Technische Universität Braunschweig

Masterarbeit

Feature-orientiertes Framing für die Verifikation von Software-Produktlinien

Autor:

Stefanie Bolle

6. Oktober 2017

Betreuer:

Prof. Dr.-Ing. Ina Schaefer

Dr.-Ing. Thomas Thüm

M.Sc. Alexander Knüppel

Institut für Softwaretechnik und Fahrzeuginformatik, TU Braunschweig

M.Sc. Stefan Krüger

Heinz Nixdorf Institut, Universität Paderborn

http://dx.doi.org/10.24355/dbbs.084-201711280920

Bolle, Stefanie:
Feature-orientiertes Framing für die Verifikation von Software-Produktlinien
Masterarbeit, Technische Universität Braunschweig, 2017.

http://dx.doi.org/10.24355/dbbs.084-201711280920

Inhaltsangabe

Software-Produktlinien sind Softwareprodukte, die eine gemeinsame Codebasis besitzen und sich in ausgewählten Features unterscheiden. Mit Feature-orientierter Programmierung wird eine Möglichkeit zur Realisierung von Produktlinien geboten. Durch den häufigen Einsatz von Produktlinien in sicherheitskritischen Systemen erweist sich die Verifikation als besonders wichtig, damit keine schwerwiegenden Fehler auftreten. Die Verifikation gestaltet sich jedoch schwierig, da jedes Produkt verifiziert und spezifiziert werden muss. Für die Spezifikation haben sich Kontrakte bewährt, wobei in der Feature-orientierten Programmierung die Komposition von den Kontrakten der Features definiert werden muss. Die Komposition von Kontrakten wurde bereits umgesetzt, aber es wurden noch nicht alle Bestandteile von Kontrakten berücksichtigt, insbesondere wird das Framing, das essentiell für die Verifikation mit Kontrakten ist, noch nicht untersucht. In der Arbeit beschäftigen wir uns damit, wie sich die Komposition von Kontrakten sinnvoll mit Framing erweitern lassen. Dazu betrachten wir Framing-Techniken in anderen Bereichen und analysieren, ob sie sich für Feature-orientiertes Framing eignen. Abschließend werden die Techniken für Feature-orientiertes Framing bezüglich ihrer Anwendbarkeit und ihres Nutzens für die Verifikation von Produktlinien evaluiert.

http://dx.doi.org/10.24355/dbbs.084-201711280920

http://dx.doi.org/10.24355/dbbs.084-201711280920

Danksagung

An dieser Stelle möchte ich mich bei meinen Betreuern Dr.-Ing. Thomas Thüm, M.Sc. Alexander Knüppel und M.Sc. Stefan Krüger bedanken, die es mir ermöglicht haben diese Arbeit umzusetzen. Ohne die engagierte Betreuung wäre ich nicht so weit gekommen. Die vielen Treffen und Diskussionen haben mich immer wieder vorwärts gebracht, wenn ich auf Probleme gestoßen bin, und die Reviews zu den Entwürfen haben mir sehr bei der stetigen Verbesserung der Arbeit geholfen.

Ein großes Dankeschön geht auch an meine Eltern, die mich während des Studiums immer unterstützt und ermutigt haben.

Zu guter Letzt möchte ich mich noch bei meinen Freunden bedanken, die mir stets zur Seite gestanden haben und für mich da waren.

http://dx.doi.org/10.24355/dbbs.084-201711280920

http://dx.doi.org/10.24355/dbbs.084-201711280920

Inhaltsverzeichnis

http://dx.doi.org/10.24355/dbbs.084-201711280920

http://dx.doi.org/10.24355/dbbs.084-201711280920

Abbildungsverzeichnis

http://dx.doi.org/10.24355/dbbs.084-201711280920

http://dx.doi.org/10.24355/dbbs.084-201711280920

Tabellenverzeichnis

http://dx.doi.org/10.24355/dbbs.084-201711280920

http://dx.doi.org/10.24355/dbbs.084-201711280920

Quelltextverzeichnis

http://dx.doi.org/10.24355/dbbs.084-201711280920

http://dx.doi.org/10.24355/dbbs.084-201711280920

1. Einführung

Software-Produktlinien werden zunehmend in der Industrie und insbesondere in sicherheitskritischen Systemen eingesetzt, da sie durch die Wiederverwendung von Softwareartefakten Zeit und Kosten bei der Entwicklung sparen [Clements and Northrop, 2001]. Eine Software-Produktlinie bezeichnet eine Menge von Softwareprodukten, die dieselbe Codebasis verwenden und sich durch ausgewählte Merkmalen, den Features, unterscheiden [Clements and Northrop, 2001, Apel et al., 2013a]. Ein typisches Beispiel hierfür sind Automobile, da hier dem Käufer mit diversen Konfigurationsmöglichkeiten eine große Produktauswahl geboten wird, wie verschiedene Assistenzsysteme oder die Farbe der Lackierung. Eine Auswahl beziehungsweise eine Menge von Features stellt die Produktkonfiguration dar, mit der aus den wiederverwendbaren Softwareartefakten das Produkt erzeugt wird [Clements and Northrop, 2001, Apel et al., 2013a].

Eine Möglichkeit zur Realisierung von Produktlinien stellt die Feature-orientierte Programmierung dar [Apel et al., 2013a]. Jedes Feature der Produktlinie wird in einem Feature-Modul implementiert, welches alle Softwareartefakte des Features enthält. Für die Produktgenerierung wird dann eine Komposition der Feature-Module aller ausgewählten Features erzeugt. Für die Komposition können verschiedene Kompositionswerkzeuge verwendet werden, zum Beispiel FeatureHouse [Apel et al., 2013b] oder AHEAD [Batory et al., 2004].

Die Verifikation von Software-Produktlinien ist speziell bei sicherheitskritischen Systemen wichtig, damit bei der Ausführung der Software keine schwerwiegenden Fehler auftreten, was im schlimmsten Fall Menschenleben gefährden kann, zum Beispiel bei der Fehlfunktion der Bremssoftware eines Fahrzeugs. Durch die enormen Produktvielfalt ist die Verifikation oftmals aufwendig, da jedes Produkt verifiziert werden muss. Die Produkte einzeln zu verifizieren ist aber redundant, da die Produkte auf gemeinsamen Merkmalen basieren. Deswegen sollten die gemeinsamen Merkmale genutzt werden, um den Verifikationsaufwand zu reduzieren. Für die Verifikation benötigen wir eine Spezifikation der Software-Produktlinie [Beckert and Hähnle, 2014],

http://dx.doi.org/10.24355/dbbs.084-201711280920

die das korrekte Verhalten der Software-Produktlinie komplett beschreibt. Durch einen Abgleich der tatsächlichen Funktion mit der Spezifikation kann dann festgestellt werden [Beckert and Hähnle, 2014], ob die Software korrekt funktioniert.

Design by Contract [Meyer, 1992, Hatcliff et al., 2012] ist eine Spezifikationsmethodik. Damit wird im Code festgelegt, wie sich die einzelnen Prozeduren semantisch verhalten, indem Vor- und Nachbedingungen für die Prozedur angegeben werden. Die Vorbedingung gibt den Zustand des Programms vor Ausführung der Prozedur an und die Nachbedingung den Zustand nach der Ausführung. Das Paar von Vor- und Nachbedingung wird als Kontrakt bezeichnet [Hatcliff et al., 2012].

Damit sich der Aufrufer auf den Kontrakt verlassen kann, muss zusätzlich der Frame der Prozedur angegeben werden, mit dem beschrieben wird, dass sich nur bestimmte Felder des Programms während der Prozedur ändern dürfen [Hatcliff et al., 2012]. Das Framing ist insbesondere für die formale Verifikation mit Kontrakten von Bedeutung [Ahrendt et al., 2016, Weiß, 2011, Kassios, 2006], da ohne Frame manche Beweise nicht geschlossen werden können. Zum Beispiel bezieht sich der Kontrakt einer Methode A auf eine Variable z und die Methode A führt eine Methode B aus. Wenn in der Nachbedingung von Methode B keine Aussage über die Variable z getroffen wird, kann nicht bestimmt werden, ob und wie sich die Variable z bei der Ausführung von Methode B geändert hat. Anders ausgedrückt heißt das, dass die Spezifikation Seiteneffekte erlaubt, wenn in der Nachbedingung keine Aussage über eine Variable getroffen wird, zum Beispiel durch Zuweisung eines Wertes [Leino and Nelson, 2002].

Die Spezifikation von Produktlinien stellt durch die Produktvielfalt eine Herausforderung dar, da es für jede valide Produktkonfiguration auch eine korrekte Spezifikation geben muss. Für Feature-orientierte Programmierung müssten wir beispielsweise festlegen, wie der Kontrakt auf die Feature-Module aufgeteilt wird beziehungsweise wie die partiellen Kontrakte aus den Feature-Modulen zu einem Kontrakt zusammengefügt werden. Es existieren schon ein paar Ansätze, wie mit Design by Contract in Kombination mit Produktlinien umgegangen werden kann. Somit haben wir für Feature-orientierte Programmierung [Thüm, 2015], Delta-orientierte Programmierung [Hähnle and Schaefer, 2012, Damiani et al., 2012, Bruns et al., 2011] und Aspekt-orientierte Programmierung [Zhao and Rinard, 2003, Rebêlo et al., 2014a, Rebêlo et al., 2014b] Ansätze, die Design by Contract für die Verifikation von Produktlinien nutzen.

Bei den Ansätzen für Desing by Contract in Produktlinien wird allerdings das Framing noch nicht hinreichend berücksichtigt. Das Framing wird entweder gar nicht betrachtet [Bruns et al., 2011, Thüm, 2015, Rebêlo et al., 2014a, Rebêlo et al., 2014b], vereinfacht indem die Kontrakte nicht redefiniert werden dürfen [Hähnle and Schaefer, 2012], es fehlt eine Komposition der Kontrakte [Damiani et al., 2012, Zhao and Rinard, 2003] oder es wird nicht auf eventuelle Probleme und Grenzen eingegangen [Hähnle et al., 2013]. Da Framing allerdings von essentieller Bedeutung ist, weil die Kontrakte ohne Framing nicht verlässlich sind [Hatcliff et al., 2012], beschäftigen

http://dx.doi.org/10.24355/dbbs.084-201711280920

wir uns in der vorliegenden Arbeit mit der Problematik, eine geeignete Technik für Feature-orientiertes Framing zu finden. Dazu befassen wir uns mit aktuellen Techniken von Framing in Objekt-orientierter Programmierung und mit den oben genannten Techniken in Produktlinien. Für die Techniken in Objekt-orientierter Programmierung überlegen wir, ob sie sich auf Feature-orientiertes Framing übertragen lassen, und für die Produktlinientechniken, ob die Ansätze sich für Feature-orientiertes Framing eignen. Außerdem betrachten wir noch unabhängig von den vorangegangenen Techniken, wie Framing für Feature-orientierte Produktlinien aussehen kann.

Zielstellung der Arbeit

Das Ziel der Arbeit besteht darin, mithilfe einer Untersuchung verschiedener Framing-Techniken und den vorhandenen Feature-orientierten Kontrakten für Vor- und Nachbedingung eine Strategie für Feature-orientiertes Framing abzuleiten. Zusätzlich wollen wir Framing in der Verifikation von Software-Produktlinien hinsichtlich seiner Nutzen und Kosten analysieren. Dazu sind folgende Schritte notwendig:

1. Analyse von Feature-orientierten Kontrakten:
   Es existieren bereits Mechanismen für die Komposition von Feature-orientierten Kontrakten [Thüm, 2015]. Anhand der Kompositionsmechanismen und ihrer Eigenschaften können wir analysieren, wie passende Framing-Techniken zu den Kompositionsmechanismen aussehen können.

2. Literaturrecherche zu Framing in der Objekt-orientierten Programmierung:
   In der Objekt-orientierten Programmierung weisen Superklassen und Subklassen ähnliche Strukturen wie Klassenverfeinerungen in Feature-orientierter Programmierung auf. Die vorhandenen Framing-Techniken für Objekt-orientierte Programmierung untersuchen wir im Hinblick auf ihre Anwendbarkeit für Feature-orientiertes Framing.

3. Evaluierung des Feature-orientierten Framings:
   Wir untersuchen 14 Fallstudien auf ihre Verwendung von Framing. Dazu betrachten wir Häufigkeit und Eigenschaften von Framing in den Produktlinien, sowie die Anwendbarkeit der entwickelten Framing-Techniken für das Feature-orientierte Framing.

4. Anwendung von Framing in der Verifikation:
   Für die Verifikation wollen wir analysieren, wie effizient die Verwendung von Framing ist. Dafür wird eine Produktlinie zweimal verifiziert. Einmal wird ein spezifischer Frame für die Methoden angegeben und einmal wird die Produktlinie ohne spezifischen Frame verifiziert, das heißt alle Felder dürfen, während der Ausführung der Methode, geändert werden.

Gliederung der Arbeit

In Kapitel 2 geben wir eine Einführung in die Grundlagen von Software-Produktlinien und Design by Contract. In Kapitel 3 befassen wir uns dann mit den aktuellen Kontraktkompositionsmechanismen nach Thüm [Thüm, 2015], welche Techniken zum

http://dx.doi.org/10.24355/dbbs.084-201711280920

Framing existieren und welche sich davon für das Feature-orientierte Framing adaptieren lassen oder wie sich das Framing alternativ umsetzen lässt. Die entwickelten Framing-Techniken evaluieren wir in Kapitel 4. Dazu analysieren wir zunächst 14 Fallstudien auf die Verwendung von Framing und die Anwendbarkeit der Framing-Techniken. Zusätzlich führen wir eine deduktive Verifikation mit Variability Encoding für eine Produktlinie mit und ohne spezifischen Frame durch, deren Ergebnisse wir anschließend vergleichen. In Kapitel 5 verweisen wir auf verwandte Arbeiten und vergleichen sie mit unseren Ergebnissen. Abschließend werden in Kapitel 6 die Ergebnisse der Arbeit zusammengefasst und wir diskutieren in Kapitel 7 zukünftige Arbeiten.

http://dx.doi.org/10.24355/dbbs.084-201711280920

2. Grundlagen

In diesem Kapitel erläutern wir die Grundlagen der beiden Hauptthemen der Arbeit. Zunächst beschäftigen wir uns in Abschnitt 2.1 mit Software-Produktlinien und stellen anschließend in Abschnitt 2.2 Kontrakte zur formalen Spezifikation und Verifikation von Software vor.

2.1 Software-Produktlinien

Eine *Software-Produktlinie* bezeichnet eine Menge von Produkten, die auf einer Menge gemeinsamer Merkmale basieren und sich in ausgewählten Merkmalen, den *Features*, unterscheiden [Clements and Northrop, 2001, Apel et al., 2013a]. Durch die gemeinsame Codebasis bieten Produktlinien ein hohes Maß an Wiederverwendung und damit Kostenreduktion an [Clements and Northrop, 2001]. Ein Beispiel dafür ist die Entwicklung von Automobilen, welche eine Vielzahl von Konfigurationen für den Käufer anbieten. Der Prozess der Entwicklung von Software-Produktlinien lässt sich generell in zwei Phasen unterteilen, dem Domain Engineering und dem Application Engineering [Clements and Northrop, 2001].

Im *Domain Engineering* wird die Software-Produktlinien als Ganzes betrachtet [Clements and Northrop, 2001, Klaus Pohl and van der Linden, 2005]. Dafür wird die Domäne analysiert und festgelegt, welche Features für die Produktlinie benötigt werden. Aus den Features wird ein *Feature-Modell* erstellt, das die validen Produktkonfigurationen enthält. Eine *Produktkonfiguration* gibt hier an, welche Features für ein Produkt ausgewählt wurden. Außerdem werden im Domain Engineering die wiederverwendbaren Softwareartefakte erstellt, zum Beispiel würden in der Feature-orientierten Programmierung die Feature-Module implementiert werden.

Im *Application Engineering* werden die produktspezifischen Eigenschaften betrachtet [Clements and Northrop, 2001, Klaus Pohl and van der Linden, 2005], das heißt alles was die einzelnen Produkte und nicht die Produktlinie als Ganzes betrifft. Für ein Produkt wird eine valide Produktkonfiguration ausgewählt, die dem Bedarf des

http://dx.doi.org/10.24355/dbbs.084-201711280920

Nutzers entspricht. Aus der Konfiguration wird mit den wiederverwendbaren Softwareartefakten aus dem Domain Engineering das Produkt generiert. Die Generierung hängt von der ausgewählten Methode, zum Beispiel Feature-orientierter oder Delta-orientierter Programmierung, ab, die entweder automatisch erfolgt oder aber noch manuelle Eingriffe, wie beispielsweise Gluecode, braucht.

2.1.1 Feature-Modellierung

Für die Software-Produktlinien wird im Domain Engineering häufig eine Übersicht über alle möglichen Featurekombinationen erstellt [Clements and Northrop, 2001,Klaus Pohl and van der Linden, 2005], da in den meisten Produktlinien nicht alle Features frei kombiniert werden dürfen. Zu diesem Zweck gibt es *Feature-Modelle*, die die valide Menge von Produktkonfigurationen repräsentieren [Kang et al., 1990].

Feature-Modelle können als aussagenlogische Formel [Batory, 2005] dargestellt werden. Dazu stellt jedes Feature eine binäre Variable dar. Wenn die Formel für die zugeordneten Werte als wahr ausgewertet wird, stellt die Belegung eine valide Konfiguration dar. Abbildung 2.1 ist ein Beispiel für eine aussagenlogische Formel für die Produktlinie *BankAccount* von Krüger [Krüger, 2014]. So würde beispielsweise nur die Auswahl der Features *BankAccount* und *InterestEstimation* dazu führen, dass die Formel als falsch ausgewertet wird und die Kombination keine valide Konfiguration ergibt.

$$
\begin{aligned}
&BankAccount \\
&\wedge (\neg\, InterestEstimation \;\vee\; Interest) \\
&\wedge (\neg\, Lock \vee\; Transaction) \\
&\wedge (\neg\, Logging \vee\; TransactionLog) \\
&\wedge (Transaction \;\wedge\; Logging \;\Leftrightarrow\; TransactionLog)
\end{aligned}
$$

Abbildung 2.1: Eine aussagenlogische Formel für die Produktlinie *BankAccount*

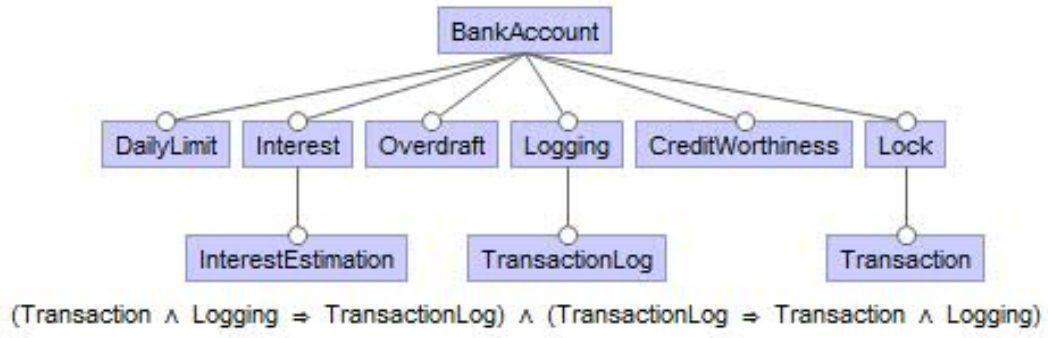

Abbildung 2.2: Ein Feature-Diagramm für die Produktlinie *BankAccount*

Da aussagenlogische Formeln schnell unübersichtlich werden, können die Feature-Modelle auch grafisch als *Feature-Diagramm* dargestellt werden [Kang et al., 1990]. Unsere aussagenlogische Formel aus Abbildung 2.1 wird mit dem Feature-Diagramm in Abbildung 2.2 repräsentiert. Alle Features der Produktlinie sind optional, aber

http://dx.doi.org/10.24355/dbbs.084-201711280920

wir haben ein paar Abhängigkeiten. Die *Eltern/Kind-Relation* stellt in diesem Kontext dar, dass das Kind-Feature nur gewählt werden darf, wenn das Eltern-Feature gewählt wurde, zum Beispiel können wir das Feature *Transaction* nur auswählen, wenn das Feature *Logging* ebenfalls verwendet wird. Zusätzlich muss bei der Auswahl die *crosstree-Constraint*, die aussagenlogische Formel unterhalb des Baums, berücksichtigt werden. Die crosstree-Constraint sagt in unserem Falle aus, dass *TransactionLog* nur gewählt werden kann, wenn *Transaction* und *Logging* gewählt sind. Die Produktlinie *BankAccount* verwenden wir in den folgenden Kapiteln als Beispiel.

2.1.2 Feature-orientierte Programmierung

Feature-orientierte Programmierung [Prehofer, 1997] beschreibt die Entwicklung von Software anhand seiner Features. Dabei werden die Softwareartefakte den verschiedenen Features zugeordnet und in einem *Feature-Modul* zusammengefasst. Die Softwareartefakte werden dann mit dem Kompositionsoperator nach Apel et al. [Apel et al., 2010] zu einem Programm komponiert (mit Programm p, einer Featuremenge F, Feature f_i):

$$\bullet : F \times F \to F$$

$$p = f_n(\bullet f_{n-1} \bullet (\cdots \bullet (f_2 \bullet f_1)))$$

Für die Komposition gibt es verschiedene Tools, die die Features auf unterschiedliche Weise komponieren, zum Beispiel FeatureHouse [Apel et al., 2013b] und AHEAD [Batory et al., 2004]. FeatureHouse nutzt für die Komposition der Softwareartefakte *Superimposition*, das heißt Softwareartefakte werden anhand der zugehörigen Unterstrukturen gemischt [Apel et al., 2013b].

Generell werden bei FeatureHouse *feature structure trees* verwendet, um den Aufbau eines Softwareartefakts zu repräsentieren, welche dann anschließend überlappt werden um die Features zu kombinieren [Apel et al., 2013b]. Für Objekt-orientierte Programmierung werden so Pakete, Klassen, Felder und Methoden als Baumstruktur dargestellt. Bei der Komposition werden dann Knoten mit gleichen Namen überlagert, es werden also neue Subknoten hinzugefügt und Subknoten mit gleichen Namen werden wieder überlagert. Für Pakete und Klassen ist das recht einfach, aber bei Feldern und Methoden nicht. Felder werden von einer Version übernommen, während Methoden kombiniert werden müssen.

Aus Implementierungssicht heißt das, dass das Feature-Modul die Klassen und Klassenverfeinerungen eines Features beinhaltet [Apel et al., 2013b]. Die Klassenverfeinerungen beschreiben, wie die Klasse eines vorherigen Feature-Moduls modifiziert wird. Dabei können Methoden und Felder den Klassen hinzugefügt werden oder vorhandene Methoden einer Klasse werden überschrieben oder mithilfe eines Schlüsselworts, für FeatureHouse ist das `original`, ergänzt. Bei der Anwendung des Schlüsselworts wird dann der Code des vorangegangenen Feature-Moduls aufgerufen.

http://dx.doi.org/10.24355/dbbs.084-201711280920

Beispiel 2.1. Betrachten wir als Beispiel unsere Produktlinie *BankAccount* aus Abbildung 2.2. Das Feature *BankAccount* beinhaltet die Klassen `Account` (Quelltext 2.1) mit den Basisfunktionen und das Feature *DailyLimit* enthält eine Klassenverfeinerung für die Klasse `Account` (Quelltext 2.2) und die neue Klasse `Application`. Die Klasse `Account` aus dem Feature *BankAccount* beinhaltet schon die Methoden `update`, `undoUpdate` und `overdraftLimitExceeded` und die Felder `balance` und `OVERDRAFT_LIMIT`. Mit der Klassenverfeinerung des Features *DailyLimit* werden die Felder `DAILY_LIMIT` und `withdraw` hinzugefügt, sowie die neue Methode `checkWithdraw`. Zusätzlich werden die vorhandenen Methoden `update` und `undoUpdate` mithilfe von `original` verfeinert. In Quelltext 2.3 sehen wir, wie die Komposition der beiden Features *BankAccount* und *DailyLimit* aussehen würde, die `original`-Schlüsselwörter werden mit einer Wrapper-Methode ersetzt, welche die Implementierung des vorherigen Features enthält.

feature module *BankAccount*

```
1  public final class Account {
2    public final int OVERDRAFT_LIMIT = 0;
3    public int balance = 0;
4    Account() {
5    }
6    boolean update(int x) {
7      int newBalance = balance + x;
8      if (!overdraftLimitExceeded(newBalance)) {
9        balance = newBalance;
10       return true;
11     }
12     return false;
13   }
14   ...
15 }
```

Quelltext 2.1: Klasse `Account` im Feature *BankAccount*

feature module *DailyLimit*

```
1  class Account {
2      public final static int DAILY_LIMIT = -1000;
3      public int withdraw = 0;
4      boolean checkWithdraw(int x, boolean undo) {
5          return (withdraw + ((undo) ? -x : x) < DAILY_LIMIT);
6      }
7      boolean update(int x) {
8          int newWithdraw = withdraw;
9          if (x < 0)  {
10             newWithdraw += x;
11             if (checkWithdraw(x, false))
12                 return false;
13         }
14         if (!original(x))
15             return false;
16         withdraw = newWithdraw;
17         return true;
18     }
19     ...
20 }
```

Quelltext 2.2: Klasse `Account` im Feature *DailyLimit*

http://dx.doi.org/10.24355/dbbs.084-201711280920

```
1  public final class Account {                          {DailyLimit,BankAccount}
2      public final int OVERDRAFT_LIMIT = 0;
3      public int balance = 0;
4      public final static int DAILY_LIMIT = -1000;
5      public int withdraw = 0;
6
7      Account() {
8      }
9
10     boolean update__wrappee__BankAccount(int x) {
11         int newBalance = balance + x;
12         if (!overdraftLimitExceeded(newBalance)) {
13             balance = newBalance;
14             return true;
15         }
16         return false;
17     }
18
19     boolean checkWithdraw(int x, boolean undo) {
20         return (withdraw + ((undo) ? -x : x) < DAILY_LIMIT);
21     }
22
23     boolean update(int x) {
24         int newWithdraw = withdraw;
25         if (x < 0)  {
26             newWithdraw += x;
27             if (checkWithdraw(x, false))
28                 return false;
29         }
30         if (!update__wrappee__BankAccount(x))
31             return false;
32         withdraw = newWithdraw;
33         return true;
34     }
35     ...
36 }
```

Quelltext 2.3: Komposition der Klasse `Account` der Features *DailyLimit* und *BankAccount*

2.1.3 Weitere Programmiertechniken für Produktlinien

Da wir in Kapitel 3 unter anderem Framing für Software-Produktlinien untersuchen wollen und uns in Kapitel 5 mit verwandten Arbeiten beschäftigen, befassen wir uns in diesem Abschnitt zunächst mit Techniken, welche Parallelen zu Feature-orientierter Programmierung aufweisen. Dafür betrachten wir zunächst Delta-orientierte Programmierung und anschließend Aspekt-orientierte Programmierung.

Delta-orientierte Programmierung bietet einem die Möglichkeit der Implementierung von Software-Produktlinien mit Deltas [Schaefer et al., 2010]. Eine Software-Produktlinien besteht aus dem Kern-Modul, den Delta-Modulen und einer Produktlinien Implementierung. Das *Kern-Modul* enthält die Codebasis, das heißt den Code, der in jeder Software-Produktlinie benötigt wird. In den *Delta-Modulen* kann dieser

http://dx.doi.org/10.24355/dbbs.084-201711280920

Code dann modifiziert, gelöscht, umbenannt oder erweitert werden. Die *Produktlinien Implementierung* beschreibt, welche Features vorhanden sind, welche Konfigurationen valide sind, welche Features zum Kern-Modul gehören, bei welcher Feature Auswahl ein Delta-Modul eingebunden werden soll und in welcher Reihenfolge die Delta-Module angewendet werden. Bei der Produktgenerierung werden die Delta-Module entsprechend der Produktlinien Implementierung auf das Kern-Modul angewandt.

Beispiel 2.2. In Quelltext 2.4 beschreiben wir das Kern-Modul `BankAccount`, in Quelltext 2.5 das Delta-Modul für das Feature *CreditWorthiness* und in Quelltext 2.6 die Produktlinien Implementierung für die Produktlinie *BankAccount* aus Abschnitt 2.1.1. Bei der Produktlinien Implementierung sehen wir unter `features` alle Features, die die Produktlinie enthält, in `configurations` steht die Bedingung für eine valide Produktlinie, unter `core` wird unser Kern-Modul `BankAccount` aufgeführt und danach folgen die Deltas der Produktlinie. Wenn wir nun ein Produkt mit dem Feature `CredithWorthiness` und keinem weiteren erzeugen wollen, wird das Delta-Modul *CredithWorthiness* auf unser Kern-Modul `BankAccount` angewandt.

```
1 core BankAccount{
2     public class Account(){
3         public final int OVERDRAFT_LIMIT = 0;
4         public int balance = 0;
5         boolean update(int x) {
6             ...
7         }
8     }
9 }
```

Quelltext 2.4: Code für das Kern-Modul vom Feature *BankAccount*

```
1 delta DCreditWorthiness when CreditWorthiness{
2     modifies class Account{
3         adds boolean credit(int amount){
4             return balance >= amount;
5         }
6     }
7 }
```

Quelltext 2.5: Delta-Modul für das Feature *CreditWorthiness*

Aspekt-orientierte Programmierung war ursprünglich als Lösung für das Problem der querschneidenden Belange gedacht [Kiczales et al., 1997]. Die *querschneidenden Belange* beschreiben, dass eine Funktion quer über das System verteilt ist, wie zum Beispiel das Protokollieren des Programmverlaufs für mehrere oder alle Methoden. Zur Kapselung wird die Funktion in einem *Aspekt* zusammengefasst. Die Aspekte ermöglichen es *Joinpoints*, die eine Stelle im Code beschreiben, im Programm anzugeben, an dem der Code (*Advice*) aus dem Aspekt eingewebt werden soll. Dabei kann Aspekt-orientierte Programmierung auch für die Entwicklung von Produktlinien verwendet werden, indem ein Aspekt ein Feature repräsentiert und die Aspekte abhängig von der Produktkonfiguration eingebunden werden.

http://dx.doi.org/10.24355/dbbs.084-201711280920

```
features BankAccount, DailyLimit, Interest, InterestEstimation,
    Overdraft, Logging, TransactionLog, CreditWorthiness,
    Lock, Transaction
configurations BankAccount &&
    (Transaction && Logging && TransactionLog)||
    ((!Transaction || !Logging) && !TransactionLog)
core BankAccount
delta DDailyLimit when DailyLimit {..}
delta DInterest when Interest {..}
delta DInterestEstimation after DInterest when InterestEstimation{..}
...
delta DCredithWothiness when CreditWorthiness{..}
...
```

Quelltext 2.6: Produktlinien Implementierung für die Produktlinie *BankAccount*

Beispiel 2.3. Betrachten wir als Beispiel Quelltext 2.7, welches AspectJ nutzt, eine Aspekt-orientierte Erweiterung für Java[1]. Dort wird mithilfe eines Aspekts das Feature *Logging* zur Klasse *Account* hinzugefügt, mit dem die einzelnen Buchungsvorgänge des Kontos protokolliert werden. Dazu wird die Methode `update` mit einem `around` Advice entsprechend angepasst und die benötigten Felder werden hinzugefügt.

```
aspect Logging{
    int[] Account.updates = new int[10];
    int Account.updateCounter = 0;
    void around(int weight) : excecution(boolean Account.update(int) {
        if (proceed(x)){
            updateCounter = (updateCounter + 1) % 10;
            updates[updateCounter] = x;
            return true;
        }
        return false;
    }
}
```

Quelltext 2.7: Aspekt für das Feature *Logging*

2.2 Design by Contract

Eine *formale Spezifikation* dient der Beschreibung des korrekten Verhaltens eines Systems. Eine Methode dafür sind *Assertions* [Hatcliff et al., 2012], die meist als seiteneffektfreie Boolesche Ausdrücke in das Programm eingebettet werden und Ausnahmen beschreiben, die nicht eintreffen dürfen. Die Assertions werden an einer beliebigen Stelle im Code platziert und können je nach Bedarf für die Verifikation genutzt werden oder auch zur Erzeugung von Fehlermeldungen im laufenden System bei Verletzung der Assertion. Optional können die Assertions als Vor- und Nachbedingung strukturiert werden. Dadurch wird für eine Prozedur mit der *Vorbedingung*

[1] https://eclipse.org/aspectj/

http://dx.doi.org/10.24355/dbbs.084-201711280920

beschrieben, was erfüllt sein muss, um nach der Ausführung der Prozedur die *Nachbedingung* zu gewährleisten. Diese Struktur wird als *Design by Contract* bezeichnet und bildet mit dem Paar der Vor- und Nachbedingung einen *Kontrakt* zwischen der Prozedur und dem *Caller* (Aufrufer der Prozedur) [Meyer, 1992, Hatcliff et al., 2012]. Design by Contract wird erstmals in EIFFEL zum Einsatz gebracht [Meyer, 1992]. Weitere Sprachen, die Design by Contract nutzen, sind JML [Leavens et al., 2006], SPEC# [Barnett et al., 2005], LARCH [Leavens, 1996] und SPARK [Barnes, 1997].

Neben Vor- und Nachbedingungen gibt es ein weiteres Konzept für die Kontrakte, das Framing [Hatcliff et al., 2012]. *Framing* beschreibt, welchen Teil des Programmzustands eine Prozedur modifizieren darf und damit implizit welchen nicht, was insbesondere für die formale Verifikation von Belangen ist [Ahrendt et al., 2016, Weiß, 2011]. Außerdem ist es ebenfalls für das Information Hiding [Leavens and Müller, 2007] und dem Vorbeugen von Seiteneffekten wichtig [Leino and Nelson, 2002].

Zusätzlich zu den Kontrakten werden in der Spezifikation auch noch *Invarianten* genutzt, welche zu bestimmten vordefinierten Zeitpunkten im Programm gelten müssen (zum Beispiel nach einer Schleifeniteration oder vor und nach jeder Methodenausführung) [Hatcliff et al., 2012].

In der Arbeit wollen wir für Design by Contract die Java Modeling Language JML [Leavens et al., 2006] verwenden, da wir für die Verifikation Java Projekte mit FEATUREHOUSE verwenden und auch KEY, ein Tool für die deduktive Verifikation [Ahrendt et al., 2016], mit JML arbeiten kann. Betrachten wir dazu ein Beispiel von Design by Contract in JML:

Beispiel 2.4. In Quelltext 2.8 ist als Beispiel der Kontrakt für die Methode `update` aus der Klasse `Account` (Quelltext 2.1) dargestellt. Dabei wird die Vorbedingung mit `requires` in Zeile 4 und die Nachbedingungen mit `ensures` in Zeile 5 und 6 angegeben. Der Frame wird mit `assignable` in Zeile 7 festgelegt. Der Ausdruck `\old(balance)` in der Nachbedingung bezieht sich auf den Wert von `balance` vor Ausführung der Methode und `\result` bezieht sich auf den Rückgabewert der Methode.

Ein paar weitere JML Konstrukte [Leavens et al., 2006] sind ebenfalls für die Thematik noch relevant:

- `\nothing` und `\everything`: Mit `assignable \nothing` wird angegeben, dass keine Felder in der Methode modifiziert werden dürfen, und mit `assignable \everything`, dass alle Felder modifiziert werden dürfen. Der Default für JML, wenn keine `assignable`-Klausel angegeben wird, ist `\everything`.
- *Spezifikationsfälle*: Die Spezifikationsfälle beschreiben mit `normal_behavior`, dass die Methode nicht mit einem Fehler oder einer Ausnahme terminiert und ihre Nachbedingung erfüllt wird, falls die Vorbedingung erfüllt ist. Dagegen beschreibt ein Spezifikationsfall mit `exceptional_behavior`, dass bei Erfüllung der Vorbedingung die Methode eine Ausnahme auslöst. Mit `also` können mehrere Spezifikationsfälle für eine Methode festgelegt werden.

http://dx.doi.org/10.24355/dbbs.084-201711280920

```
1  public final class Account {
2      ...
3      /*@
4        @ requires x != 0;
5        @ ensures \result ==> \old(balance) + x >= OVERDRAFT_LIMIT
6              && \old(balance) + x < OVERDRAFT_LIMIT ==> !\result;
7        @ assignable balance;
8        @*/
9      boolean update(int x) {
10         int newBalance = balance + x;
11         if (!overdraftLimitExceeded(newBalance)) {
12             balance = newBalance;
13             return true;
14         }
15         return false;
16     }
17     ...
18 }
```

Quelltext 2.8: JML Kontrakt für die Methode `update` der Klasse `Account`

- *Invarianten*: Die Invarianten werden mit `invariant` dargestellt und dürfen nur während der Ausführung einer Methode verletzt werden.
- *model fields*: model fields stellen Variablen dar, die nur für die Spezifikation verwendet werden. Mit `model` können sie in JML deklariert werden und mit `represents` wird der Wert der Variable beschrieben.
- *pure methods*: Java Methoden können als `pure` deklariert werden, was bedeutet, dass die Methode keine Seiteneffekte hat beziehungsweise der Frame `assignable \nothing` ist. Die pure methods können dann in Invarianten, Vor- und Nachbedingungen eingesetzt werden.

http://dx.doi.org/10.24355/dbbs.084-201711280920

http://dx.doi.org/10.24355/dbbs.084-201711280920

3. Feature-orientiertes Framing

In diesem Kapitel untersuchen wir, wie Feature-orientiertes Framing aussehen muss, um in Feature-orientierten Kontrakten Anwendung zu finden. Für Kontrakte in der Feature-orientierten Programmierung müssen wir die Kontrakte aus den einzelnen Features komponieren. Die Kontraktkomposition existiert bereits für die Feature-orientierte Programmierung und wurde mit sechs Kontraktkompositionsmechanismen von Thüm [Thüm, 2015] umgesetzt. Diese unterstützen die Komposition von Vor- und Nachbedingung in Kontrakten, aber nicht das Framing. Das Framing ist allerdings essentieller Bestandteil für die Verifikation mit Kontrakten, denn einige Beweise lassen sich ohne Aussage über die Änderung von Feldern nicht durchführen, wie wir im Beispiel 3.1 sehen. Da wir außerdem eine Vielzahl von Produkten in der Feature-orientierten Programmierung verifizieren müssen, ist es essentiell wichtig, die Beweise wiederzuverwenden, da ansonsten die Verifikation nicht skaliert. Problematisch wird dies beim Framing, da bestimmte Eigenschaften und damit Einschränkungen für den Frame gelten müssen, um modulare Beweise durchführen zu können, die wir zusammen mit den Eigenschaften für Kontraktkomposition erläutern werden.

Beispiel 3.1. In Quelltext 3.1 sehen wir einen Ausschnitt aus dem Feature *DailyLimit* der Klasse `Account` von der Produktlinie *BankAccount*. Wir wollen hier mit Hilfe der Kontrakte die Methode `update` beweisen. Das Feld `withdraw` steht in der Nachbedingung und soll einen neuen Wert annehmen. Da wir uns bei der Verifikation mit Kontrakten nur auf den Kontrakt beziehen, erhalten wir nach der Ausführung der Methode `checkWithdraw` deren Nachbedingung als Garantie. Das Problem ist allerdings, dass in der Nachbedingung keine Aussage über `withdraw` getroffen wird, weshalb der Wert von `withdraw` nach Ausführung von `checkWithdraw` unklar ist und wir deshalb die Nachbedingung von `update` nicht beweisen können. Deshalb benötigen wir hier den Frame zum Schließen des Beweises, damit der Kontrakt von `checkWithdraw` angeben kann, dass `withdraw` nicht geändert wurde.

In Abschnitt 3.1 stellen wir zunächst die Eigenschaften für Kontraktkomposition vor, gehen auf die Kontraktkompositionsmechanismen ein und bestimmen anhand

http://dx.doi.org/10.24355/dbbs.084-201711280920

```
1  class Account {                                          feature module DailyLimit
2      /*@ requires x < 0;
3        @ ensures \result == ((!undo ==> (withdraw + x < DAILY_LIMIT))
4        @ && (undo ==> (withdraw - x < DAILY_LIMIT)));
5        @*/
6      boolean checkWithdraw(int x, boolean undo) {
7          return (withdraw + ((undo) ? -x : x) < DAILY_LIMIT);
8      }
9      /*@
10       @ requires true;
11       @ ensures \result ==> (((x < 0) ==>
12       @ (\old(withdraw) + x >= DAILY_LIMIT))
13       @ && \old(\original(x)));
14       @ ensures (((x < 0) && (\old(withdraw) + x < DAILY_LIMIT))
15       @ || !\old(\original(x))) ==> !\result;
16       @*/
17     boolean update(int x) {
18         int newWithdraw = withdraw;
19         if (x < 0)  {
20             newWithdraw += x;
21             if (checkWithdraw(x, false))
22                 return false;
23         }
24         if (!original(x))
25             return false;
26         withdraw = newWithdraw;
27         return true;
28     }
29 }
```

Quelltext 3.1: Ausschnitt aus dem Feature *DailyLimit*

ihrer Eigenschaften, wie Framing für die einzelnen Mechanismen spezifiziert werden muss. Da das so entwickelte Framing nicht immer ausreicht, befassen wir uns in Abschnitt 3.2 mit Framing in der Objekt-orientierten Programmierung und wie wir diese Techniken auf Feature-orientierte Programmierung anwenden können und welche Probleme es bei der Anwendung gibt. Abschließend fassen wir unsere Ergebnisse in Abschnitt 3.3 zusammen.

3.1 Framing für Feature-orientierte Kontraktkomposition

Eine Komposition von Kontrakten liefert Thüm [Thüm, 2015] mit den *Feature-orientierten Kontrakten*, deren Funktionsweise und Eigenschaften wir in diesem Kapitel vorstellen. Dafür befassen wir uns zunächst mit den Eigenschaften von Kontrakten und Kontraktkompositionen, die wir mit Framing ergänzen. Anschließend werden die sechs Kompositionsmechanismen vorgestellt und eine erste Erweiterung mit Framing vorgenommen. Abschließend geben wir noch einen Überblick über alle Mechanismen.

Für die Erläuterung der Kontraktkompositionsmechanismen nutzen wir ein Beispiel aus der Produktlinie *BankAccount* von Krüger [Krüger, 2014] (siehe Abbildung 2.2),

http://dx.doi.org/10.24355/dbbs.084-201711280920

das wir zur Veranschaulichung der Kontraktkomposition abgeändert haben. In Quelltext 3.2 befindet sich die Methode `update` des Features *BankAccount*, die einem Kontoguthaben `balance` den Betrag `x` hinzufügt oder abhebt. Dafür haben wir als Vorbedingung, dass `x` nicht 0 sein darf, und als Nachbedingung erhalten wir als neues Guthaben `\old(balance) + x`. Deshalb ist nur das Feld `balance` im Frame. Im Feature *DailyLimit* in Quelltext 3.3 wird die Methode verfeinert, indem ein tägliches Abhebungslimit hinzugefügt wird. Dafür möchten wir der Methode eine Vorbedingung und eine Nachbedingung hinzufügen und den Frame erweitern. Die Vorbedingung soll festlegen, dass das tägliche Abhebungslimit `DAILY_LIMIT` nicht im Vorfeld von der täglichen Abhebung `withdraw` überschritten wurde, welches den Gesamtbetrag, der an einem Tag abgehoben beziehungsweise eingezahlt worden ist, beinhaltet. Die Nachbedingung soll sicher stellen, dass die tägliche Abhebung `withdraw` aktualisiert worden ist oder bei Überschreitung des Limits `false` zurückgegeben wird. Dementsprechend muss der Frame das Feld `withdraw` enthalten. Jetzt stellt sich die Frage, wie wir den Kontrakt spezifizieren können?

feature module *BankAccount*

```
1  class Account {
2      /*@ requires x != 0;
3        @ ensures \result ==> balance == \old(balance) + x ;
4        @ assignable balance;
5        @*/
6      boolean update(int x) {
7          balance = balance + x;
8          return true;
9      }
10     ...
11 }
```

Quelltext 3.2: JML Kontrakt für die Methode `update` der Klasse `Account` im Feature *BankAccount*

3.1.1 Eigenschaften von Kontraktkomposition

In diesem Abschnitt erläutern wir zunächst die Eigenschaften von Kontrakten und Kontraktkompositionen und erweitern die Eigenschaften mit Framing, da sie bisher nur Vor- und Nachbedingung berücksichtigen. Die Eigenschaften wollen wir bei den Kompositionsmechanismen bewahren, wenn wir dort Framing ergänzen, da die Eigenschaften Vorteile für die Verifikation bieten [Thüm, 2015].

Um über die Eigenschaften der Kontraktkomposition sprechen zu können, stellen wir zunächst die verwendete Notation nach Thüm [Thüm, 2015] vor (siehe Tabelle 3.1) und ergänzen darin das Framing. Einen Kontrakt stellen wir als *Hoare-Tripel* [Hoare, 1969] mit $c = \{\phi\}\, m\, \{\psi\}$ dar, das mit ϕ die Vorbedingung der Methode m und mit ψ die Nachbedingung ausdrückt. Ein komponierter Kontrakt von dem Originalkontrakt c und der Kontraktverfeinerung $c' = \{\phi'\}\, m'\, \{\psi'\}$ wird mit $c'' = c' \bullet_M c = \{\phi\}\, m\, \{\psi\} \bullet_M \{\phi'\}\, m\, \{\psi'\} = \{\phi''\}\, m \bullet m'\, \{\psi''\}$ ausgedrückt. Um zu bestimmen, wie ϕ'' und ψ'' aussehen, wird ein Kontraktkompositionsmechanismus M genutzt, sodass sich als Kompositionsoperator $\bullet_M : C \times C \to C$ ergibt, wobei C die Menge aller gültigen Kontrakte in einer kontraktspezifischen Sprache, wie zum

http://dx.doi.org/10.24355/dbbs.084-201711280920

```
1  class Account {                                          feature module DailyLimit
2      public final static int DAILY_LIMIT = −1000;
3      public int withdraw = 0;
4      /*@ requires ?
5        @ ensures ?
6        @ assignable ?
7        @*/
8      boolean update(int x) {
9          int newWithdraw = withdraw;
10         if (x < 0)  {
11             newWithdraw += x;
12             if (newWithdraw < DAILY_LIMIT)
13                 return false;
14         }
15         withdraw = newWithdraw;
16         original(x);
17     }
18     ...
19 }
```

Quelltext 3.3: JML Kontrakt für die Methode `update` der `Account` Klasse im Feature *DailyLimit*

Beispiel JML, repräsentiert.

Da in der Notation die Änderung des Frames noch nicht dargestellt wird, erweitern wir die Notation zu $c = \{\phi\}\, m\, \{\psi\}\, [\alpha]$ mit dem Frame α. Dementsprechend wird die Kontraktkomposition mit $c'' = c' \bullet_M c = \{\phi\}\, m\, \{\psi\}\, [\alpha] \bullet_M \{\phi'\}\, m\, \{\psi'\}\, [\alpha'] = \{\phi''\}\, m \bullet m'\, \{\psi''\}\, [\alpha'']$ dargestellt, sodass der Kontraktkompositionsmechanismus nun auch definieren muss, wie die Komposition der Frames α und α' gehandhabt wird. Wenn wir nicht explizit eine andere Notation angeben, wird die hier erläuterte Notation verwendet.

Generell können auch mehr als zwei Kontrakte komponiert werden, also $c_{comp} = c_n \bullet_M c_{n-1} \cdots \bullet_M c_1$. Da wir aber die Komposition von mehreren Kontrakten auch in Kompositionen von je zwei Kontrakten zerlegen können, reden wir hier nur über die Komposition von zwei Kontrakten.

Für den Kontrakt gibt es zwei Sichten, den *Caller* und den *Callee* [Thüm, 2015]. Der Caller ruft eine Methode auf und der Callee ist die Methode, die aufgerufen wird. Der Caller muss also die Vorbedingung des Callees erfüllen und bekommt dafür die Nachbedingung des Callees garantiert. Dementsprechend bekommt der Callee seine Vorbedingung garantiert und muss die Nachbedingung seines Kontrakts erfüllen.

Für die Kontraktverfeinerungen wird die Kompatibilität bezüglich der Caller und Callees definiert [Thüm, 2015]. Eine Kontraktverfeinerung c' gilt als *Caller-kompatibel* zu einem Originalkontrakt c, wenn $\phi \models \phi'$ und $\psi' \models \psi$ gilt. Als *Callee-kompatibel* gilt eine Kontraktverfeinerung, die $\phi' \models \phi$ und $\psi \models \psi'$ erfüllt. Vorteil bei

http://dx.doi.org/10.24355/dbbs.084-201711280920

Bedeutung	Notation
Methode	m, m', m''
Vorbedingung	ϕ, ϕ', ϕ''
Nachbedingung	ψ, ψ', ψ''
Kontrakt	$c = \{\phi\}\, m\, \{\psi\}$
Kontraktkomposition	$c'' = c' \bullet_M c = \{\phi''\}\, m \bullet m'\, \{\psi''\}$
Kontraktkompositionsmechanismus	M
Kontraktkompositionsoperator	$\bullet_M : C \times C \to C$
Frame	α, α', α''
Kontrakt mit Frame	$c = \{\phi\}\, m\, \{\psi\}\, [\alpha]$
Kontraktkomposition mit Frame	$c'' = c' \bullet_M c = \{\phi''\}\, m \bullet m'\, \{\psi''\}\, [\alpha'']$

Tabelle 3.1: Notation für die Kontraktkomposition

einem Caller-kompatiblen Kontrakt ist, dass sich alle Caller, die sich auf c verlassen, auch auf c' verlassen können. Dementsprechend gilt für Callee-kompatible Kontrakte, dass die Callees nicht über die Kontraktänderung Bescheid wissen müssen. Im Kontext von der Feature-orientierten Programmierung ist das wichtig, da viele Features optional sind und im endgültigen Produkt beziehungsweise der endgültigen Spezifikation nicht vorhanden sind.

Basierend auf unserer Notation mit dem Frame des Kontrakts müssen wir die Caller-Kompatibilität und die Callee-Kompatibilität ebenfalls ergänzen, da der Frame für die Eigenschaften von Kontraktkompositionen relevant ist. Für die Überlegung, welche Bedingung der Frame erfüllen muss, können wir die Nachbedingung des Kontrakts betrachten, da der Frame auch als Konjunktion von $\backslash old(x) == x$ für alle nicht geänderten Felder in der Nachbedingung dargestellt werden kann. Für Caller-Kompatibilität von c' zu c gilt $\psi' \models \psi$ und da die Nachbedingungen aus Konjunktionen der nicht änderbaren Variablen steht, darf c' den Frame von c nur verkleinern. Damit erhalten wir als Bedingung $\alpha' \subseteq \alpha$ für die Caller-Kompatibilität. Analog folgt für die Callee-Kompatibilität $\alpha \subseteq \alpha'$.

Darauf aufbauend sind die *Preserving-Eigenschaften* der Kontraktkomposition bezüglich des Originalkontrakts und der Kontraktverfeinerung für $c'' = c' \bullet_M c$ wie folgt definiert [Thüm, 2015], wobei sich die Caller- und Callee-Kompatibilität auf unsere Erweiterung mit Framing bezieht:

- *original-caller-preserving*: $\forall c, c'$ ist c'' Caller-kompatibel zu c
- *refinement-caller-preserving*: $\forall c, c'$ ist c'' Caller-kompatibel zu c'
- *original-callee-preserving*: $\forall c, c'$ ist c'' Callee-kompatibel zu c
- *refinement-callee-preserving*: $\forall c, c'$ ist c'' Callee-kompatibel zu c'
- *original-preserving*: c'' ist original-preserving, wenn c'' original-caller-preserving und original-callee-preserving ist

http://dx.doi.org/10.24355/dbbs.084-201711280920

- *refinement-preserving*: c'' ist refinement-preserving, wenn c'' refinement-caller-preserving und refinement-callee-preserving ist
- *caller-preserving*: c'' ist caller-preserving, wenn c'' original-caller-preserving und refinement-caller-preserving ist
- *callee-preserving*: c'' ist callee-preserving, wenn c'' original-callee-preserving und refinement-callee-preserving ist

Mit den kombinierten Preserving-Eigenschaften (original-preserving, refinement-preserving, caller-preserving, callee-preserving) werden verschiedene Formen des *Modular Reasoning* unterstützt [Thüm, 2015]. Mit original-preserving wird der Kontrakt nur durch einen äquivalenten Kontrakt ersetzt, wodurch sich Caller und Callee auf den Kontrakt verlassen können, unabhängig von weiteren folgenden Modulen. Beim refinement-preserving ist es andersherum, Caller und Callee können sich auf den Kontrakt verlassen unabhängig von vorherigen Modulen. Caller-preserving ermöglicht Modular Reasoning für die Caller, damit kann sich ein Caller auf die Kontrakte, die in dem gleichen Modul definiert wurden verlassen. Gleiches gilt beim callee-preserving für die Callees, die Methode muss nur ihren Kontrakt erfüllen und nicht den aus vorherigen oder späteren Modulen.

Weiterhin können die Kontraktkompositionsmechanismen kommutativ, assoziativ und idempotent sein [Thüm, 2015]. Ein Kompositionsmechanismus ist *kommutativ*, wenn die Reihenfolge der Kontrakte irrelevant ist ($c \bullet_M c' = c' \bullet_M c$). Bei mehr als zwei Kontrakten gilt der Kompositionsmechanismus als *assoziativ*, falls die Reihenfolge der Ausführung der Verknüpfung beliebig sein kann ($(c \bullet_M c') \bullet_M c'' = c \bullet_M (c' \bullet_M c'')$). Für die Komposition von zwei identischen Kontrakten ist ein Kompositionsmechanismus *idempotent*, wenn der daraus resultierende Kontrakt äquivalent zu dem Originalkontrakt ist ($c \bullet_M c \equiv c$).

3.1.2 Plain Contracting

Plain Contracting verbietet das Verfeinern eines Kontrakts, der Originalkontrakt bleibt erhalten [Thüm, 2015]:

$$c' \bullet_{PC} c = \{\phi\}\, m' \bullet m\, \{\psi\}$$

Die Idee dahinter ist, dass der Aufwand für die Spezifikation geringer ist und die Verifikation einfach ist. Das liegt an den Kontrakten, die, unabhängig von der Featureauswahl, für die Methoden immer gleich bleiben, es sei denn es existiert eine alternative Methodeneinführung mit einem anderen Kontrakt, zum Beispiel bei mehreren optionalen Features. Dafür lässt sich aber kein neues Verhalten spezifizieren, wodurch der Kontrakt unzuverlässig wird und der Originalkontrakt von jeder Methodenverfeinerung erfüllt werden muss. Plain Contracting ist original-preserving, idempotent und assoziativ, aber nicht kommutativ [Thüm, 2015].

http://dx.doi.org/10.24355/dbbs.084-201711280920

Plain Contracting ist original-preserving, deshalb müssen wir die Bedingung für den Frame ebenfalls gewährleisten, deshalb gilt $\alpha'' \subseteq \alpha$ wegen der original-caller-preserving Eigenschaft und $\alpha \subseteq \alpha''$ wegen der original-callee-preserving Eigenschaft, also muss $\alpha'' = \alpha$ für den neuen Frame α'' erfüllt sein. Der Frame darf dementsprechend nicht verändert werden und es ergibt sich für die Kontraktkomposition:

$$c' \bullet_{PC} c = \{\phi\}\, m' \bullet m\, \{\psi\}\, [\alpha]$$

Die Eigenschaften bleiben erhalten. Die Idempotenz folgt mit $\alpha \bullet_{PC} \alpha = \alpha \equiv \alpha$.
Und die Assoziativität ergibt sich mit:
$(\alpha'' \bullet_{PC} \alpha') \bullet_{PC} \alpha = \alpha' \bullet_{PC} \alpha = \alpha = \alpha'' \bullet_{PC} \alpha = \alpha'' \bullet_{PC} (\alpha' \bullet_{PC} \alpha)$
Neues Verhalten können wir weiterhin nicht spezifizieren, aber durch einen unverändert Frame wird zugesichert, dass sich keine neuen Felder ändern. Das bedeutet gegenüber dem ursprünglichen Plain Contracting können wir uns jetzt auf den Kontrakt verlassen, da durch den Frame die nicht enthaltenen Variablen nicht geändert werden können ohne den Kontrakt zu verletzen, was bei der Version ohne Framing noch möglich war. Dafür begrenzt der Frame so die Implementierungsmöglichkeit von den Methodenverfeinerungen.

Beispiel 3.2. Wenn Plain Contracting auf Quelltext 3.2 und Quelltext 3.3 angewandt wird, wird der Kontrakt von Quelltext 3.2 übernommen und wir müssen in Quelltext 3.3 keinen Kontrakt spezifizieren. Das Problem ist, dass dadurch die Implementierung in Quelltext 3.2 nicht der Spezifikation entspricht, da das Feld `withdraw` im Frame fehlt. Angenommen `withdraw` wäre im Frame, dann könnten Caller sich trotzdem nicht auf den Kontrakt verlassen, da die Änderung des Feldes `withdraw` nicht spezifiziert ist. Die Bedingungen und den Frame schon im ersten Feature anzugeben ist auch keine Option, da wir so zum einen den Code von gegebenenfalls optionalen Features im Kontrakt haben, der bei nicht gewähltem Feature nicht gebraucht wird, und zum anderen sind die Felder in dem Feature teilweise noch gar nicht deklariert, was zu Fehlern in der Spezifikation führt.

3.1.3 Contract Overriding

Beim *Contract Overriding* wird der Originalkontrakt mit der Kontraktverfeinerung vollständig überschrieben [Thüm, 2015]:

$$c' \bullet_{CO} c = \{\phi'\}\, m' \bullet m\, \{\psi'\}$$

Gegenüber dem Plain Contracting wird das Verfeinern eines Kontrakts ermöglicht. Allerdings muss die Spezifikation des vorherigen Kontraktes gegebenenfalls kopiert werden, was bei Änderungen zu Inkonsistenzen führen kann. Zudem besteht bei mehr als zwei optionalen Verfeinerungen für eine Methode das Problem, dass entweder die Spezifikation im darauf folgenden Feature wiederholt wird, wodurch sie unzutreffend wird, wenn das Feature nicht gewählt ist, oder sie wird nicht wiederholt und fehlt dementsprechend, wenn das Feature gewählt ist. Außerdem müssen die Caller alle Kontrakte kennen, da sie im Vorfeld nicht wissen, auf welchen Kontrakt sie sich verlassen können. Contract Overriding ist refinement-preserving und genau wie Plain

http://dx.doi.org/10.24355/dbbs.084-201711280920

Contracting idempotent und assoziativ, aber nicht kommutativ [Thüm, 2015].

Um den Eigenschaften weiterhin zu entsprechen, muss die Frame-Technik für Contract Overriding refinement-preserving sein, das heißt wegen der Eigenschaften refinement-caller-preserving und refinement-callee-preserving gilt für den komponierten Frame $\alpha'' = \alpha'$. Dadurch lässt sich der Frame genauso überschreiben, wie der Rest des Kontrakts:

$$c' \bullet_{CO} c = \{\phi'\}\, m' \bullet m\, \{\psi'\}\, [\alpha']$$

Die restlichen Eigenschaften bleiben wie beim Plain Contracting erhalten, wobei die Begründung analog zu Plain Contracting ist, weil der Frame hier das Komplement zum Frame im Plain Contracting darstellt. Außerdem muss, genau wie für die Vor- und Nachbedingungen, der Frame wiederholt werden, damit die Spezifikation korrekt ist. Aber dadurch haben wir die gleiche Problematik mit Kopien, wie beim Rest des Kontrakts.

Beispiel 3.3. Für die Anwendung von Contract Overriding auf Quelltext 3.2 und Quelltext 3.3 muss der Kontrakt von Quelltext 3.3 spezifiziert werden. Dafür müssen die Vorbedingung, die Nachbedingung und der Frame aus Quelltext 3.2 im Kontrakt ergänzt werden, wodurch sich als neuer Kontrakt der Kontrakt in Quelltext 3.4 ergibt. Problematisch wären die Änderungen aber nur, wenn das Feature *BankAccount* optional wäre, dann würde die Nachbedingung $\backslash result ==> \backslash old(balance) + x$ nicht funktionieren, weil das Feld `balance` im Feature *BankAccount* definiert wurde und so die Spezifikation fehlerhaft wäre.

feature module *DailyLimit*

```
1  class Account {
2      ...
3      /*@ requires x != 0, withdraw >= DAILY_LIMIT;
4        @ ensures (\result ==> withdraw == \old(withdraw) + x) &&
5          (\old(withdraw) + x < DAILY_LIMIT ==> !\result) &&
6           \result ==> balance == \old(balance) + x;
7        @ assignable withdraw,balance;
8        @*/
9      boolean update(int x) {...}
10     ...
11 }
```

Quelltext 3.4: Kontrakt der Methode `update` im Feature *DailyLimit* für Contract Overriding

3.1.4 Explicit Contract Refinement

Explicit Contract Refinement soll den Nachteil der Kopien von Contract Overriding ausgleichen, indem die Vor- und Nachbedingungen der vorherigen Verfeinerung mit dem Schlüsselwort `original` eingebunden werden können [Thüm, 2015]:

$$c' \bullet_{ECR} c = \{\phi' \bullet \phi\}\, m' \bullet m\, \{\psi' \bullet \psi\}$$

http://dx.doi.org/10.24355/dbbs.084-201711280920

Mit $\phi' \bullet \phi$ wird beschrieben, dass in ϕ' `original` durch ϕ ersetzt wird, analog läuft es für ψ ab. Explicit Contract Refinement kann als Contract Overriding benutzt werden, indem `original` nicht verwendet wird. Mit Explicit Contract Refinement können also Vor- und Nachbedingungen wiederverwendet, verfeinert und überschrieben werden. Das Problem für die Caller, dass sie alle Kontrakte kennen müssen, bleibt damit aber bestehen. Zusätzlich bestehen gegebenenfalls tote Referenzen, das heißt, es wird `original` verwendet, obwohl in keinem vorherigen Feature ein Kontrakt spezifiziert wurde, wodurch ein Fehler in der Spezifikation entsteht. Das Problem besteht allerdings auch in den Methoden, wenn dort mit `original` eine noch nicht definierte Methode aufgerufen wird. Explicit Contract Refinement ist assoziativ, aber weder idempotent noch kommutativ [Thüm, 2015].

Explicit Contract Refinement wurde schon für Delta-orientierte Programmierung von Hähnle et al. [Hähnle et al., 2013] eingeführt. Dazu werden *Contract Deltas* erstellt, bei denen die Kontrakte mithilfe von `modifies` und `adds also` angepasst werden können. Mit `modifies` kann `requires`, `ensures` und `assignable` geändert werden und mithilfe von `original` die Klausel aus dem vorherigen Basiscode oder Delta aufgerufen werden. Zusätzlich kann mit `adds also` ein neuer Spezifikationsfall hinzugefügt werden. Gegenüber dem Feature-orientierten Explicit Contract Refinement haben wir in den Contract Deltas durch das `original` schon eine Komposition für Framing.

Für den Frame im Explicit Contract Refinement wollen wir wieder die Eigenschaften bewahren, was in diesem Fall nur die Assoziativität ist. Außerdem sollte es möglich sein den Frame wiederzuverwenden, zu verfeinern und zu überschreiben, da wir dies für die Vor- und Nachbedingung ebenfalls ermöglichen. Eine Option ist es analog zur Delta-orientierten Programmierung für den Frame `original` einzuführen:

$$c' \bullet_{ECR} c = \{\phi' \bullet \phi\}\, m' \bullet m\, \{\psi' \bullet \psi\}\, [\alpha' \bullet \alpha]$$

Dafür wird, genau wie bei ϕ und ψ, im komponierten Frame $\alpha' \bullet \alpha$ der Aufruf von `original` in dem verfeinertem Frame α' durch den Originalframe α ersetzt. Dadurch bleibt auch die Assoziativität erhalten, die sich analog zu den Beweisen für ϕ und ψ von Thüm [Thüm, 2015] zeigen lässt.

Beispiel 3.4. Auch bei Explicit Contract Refinement bei Quelltext 3.2 und Quelltext 3.3 muss der Kontrakt von Quelltext 3.3 spezifiziert werden. Dafür muss zusätzlich zu der neuen Vorbedingung, der neuen Nachbedingung und der Ergänzung im Frame in Quelltext 3.3 das `original` im Kontrakt angegeben werden, wodurch sich als neuer Kontrakt für das Feature *DailyLimit* der Kontrakt in Quelltext 3.5 ergibt. Anschließend kann Explicit Contract Refinement angewendet werden, wodurch sich der Kontrakt in Quelltext 3.6 ergibt. Das Problem mit den Kopien aus Contract Overriding besteht nicht mehr, sollte das Feature *BankAccount* nicht ausgewählt sein, würde das Feature vor *BankAccount* bei `original` eingebunden werden. Falls allerdings kein Feature vor *DailyLimit* existiert, hat der Kontrakt eine tote Referenz und es kommt zu einem Fehler.

http://dx.doi.org/10.24355/dbbs.084-201711280920

```
class Account {                                          feature module DailyLimit
    ...
    /*@ requires withdraw >= DAILY_LIMIT, \original;
      @ ensures (\result ==> withdraw == \old(withdraw) + x) &&
        (\old(withdraw) + x < DAILY_LIMIT ==> !\result) &&
        \original;
      @ assignable withdraw, \original;
      @*/
    boolean update(int x) {...}
    ...
}
```

Quelltext 3.5: Kontrakt für die Methode `update` im Feature *DailyLimit* für Explicit Contract Refinement

```
public final class Account {                             composition ECR
    ...
    /*@ requires withdraw >= DAILY_LIMIT, x != 0;
      @ ensures (\result ==> withdraw == \old(withdraw) + x) &&
            (\old(withdraw) + x < DAILY_LIMIT ==> !\result) &&
            \result ==> balance == \old(balance) + x ;
      @ assignable withdraw, balance;
      @*/
    boolean update(int x){...}
    ...
}
```

Quelltext 3.6: Komposition mit Explicit Contract Refinement

3.1.5 Conjunctive Contract Refinement

Im *Conjunctive Contract Refinement* werden die Vor- und Nachbedingungen mit Konjunktion verknüpft [Thüm, 2015]. Dies hat den Vorteil, dass wir implizit die Vor- und Nachbedingung des Originalkontrakts und der Kontraktverfeinerung wiederverwenden:

$$c' \bullet_{ConjCR} c = \{\phi' \wedge \phi\}\, m' \bullet m\, \{\psi' \wedge \psi\}$$

Dadurch gibt es keinen weiteren Spezifikationsaufwand für die Komposition. Es müssen nur die Vor- und Nachbedingungen für Originalkontrakt und Kontraktverfeinerung angegeben werden. Allerdings können nur Vor- und Nachbedingungen hinzugefügt werden, das Löschen von Teilen oder der gesamten Bedingungen ist nicht möglich. Durch die Konjunktion müssen die Caller alle Vorbedingungen erfüllen und die Kontrakte im Vorfeld kennen, wodurch kein Modular Reasoning möglich ist. Außerdem ergeben sich aus den Eigenschaften von Konjunktion, dass Conjunctive Contract Refinement assoziativ, idempotent und kommutativ ist [Thüm, 2015].

Für den Frame ziehen wir die Überlegung heran, dass der Frame auch in der Nachbedingung als Konjunktion mit $\backslash old(x) == x$ für alle Felder x, die nicht geändert werden dürfen, ausgedrückt werden kann. Da die Nachbedingungen der beiden Kontrakte bei Conjunctive Contract Refinement mit Konjunktion verknüpft werden,

http://dx.doi.org/10.24355/dbbs.084-201711280920

müssten wir alle Felder, die entweder im Originalkontrakt oder der Kontraktverfeinerung nicht geändert werden, zur Nachbedingung des komponierten Kontrakts hinzufügen. Damit erhalten wir im Frame die Schnittmenge vom Originalframe und dem verfeinerten Frame, was wir im Folgenden als Frame Cut bezeichnen:

$$c' \bullet_{ConjCR} c = \{\phi' \wedge \phi\}\, m' \bullet m\, \{\psi' \wedge \psi\}\, [\alpha' \cap \alpha]$$

Da der Schnitt von Mengen ebenfalls assoziativ, idempotent und kommutativ ist, bleiben die Eigenschaften erhalten.

Beispiel 3.5. Quelltext 3.7 zeigt den Kontrakt für `update` im Feature *DailyLimit*. Die Vor- und Nachbedingung können wir wie gewohnt angeben, aber der Frame ist nicht ganz so einfach. Hier haben wir uns dafür entschieden, `balance` und `withdraw` in den Frame aufzunehmen. Dadurch erhalten wir bei Komposition der Features in Quelltext 3.8 allerdings nur `balance` im Frame, wodurch wir `withdraw` nicht modifizieren dürften. Das heißt, wenn ein neues Feld modifiziert werden soll, muss es in beiden Features angegeben werden, was in diesem Beispiel jedoch nicht funktioniert, da `withdraw` im Feature *BankAccount* noch nicht deklariert wurde. Eine andere Möglichkeit wäre es, in *BankAccount* den Frame von `update` mit `\everything` anzugeben und in *DailyLimit* wieder wie in Quelltext 3.8, dann hätten wir in der Komposition einen korrekten Frame, aber wenn wir *DailyLimit* nicht ausgewählt haben, ist der Frame für `update \everything` und wäre damit zu unspezifisch.

feature module *DailyLimit*

```
1  class Account {
2      ...
3      /*@ requires withdraw >= DAILY_LIMIT;
4        @ ensures (\result ==> withdraw == \old(withdraw) + x) &&
5          (\old(withdraw) + x < DAILY_LIMIT ==> !\result);
6        @ assignable balance, withdraw;
7        @*/
8      boolean update(int x) {...}
9      ...
10 }
```

Quelltext 3.7: Kontrakt für die Methode `update` im Feature *DailyLimit* für Conjunctive Contract Refinement

3.1.6 Cumulative Contract Refinement

Das *Cumulative Contract Refinement* ermöglicht das Modular Reasoning im Gegensatz zu Conjunctive Contract Refinement [Thüm, 2015]. Dies wird durch das Prinzip von Meyer [Meyer, 1988] ermöglicht, welches angibt, dass die Vorbedingung nur geschwächt werden und die Nachbedingung nur gestärkt werden darf. Dafür wird die Vorbedingung mit Disjunktion und die Nachbedingung mit Konjunktion verknüpft:

$$c' \bullet_{CumCR} c = \{\phi' \vee \phi\}\, m' \bullet m\, \{\psi' \wedge \psi\}$$

Cumulative Contract Refinement ist caller-preserving, wodurch das Modular Reasoning bezüglich der Caller ermöglicht wird. Im Gegenzug ist der Kontrakt für

http://dx.doi.org/10.24355/dbbs.084-201711280920

```
1  public final class Account {                                    composition ConjCR
2      ...
3      /*@ requires withdraw >= DAILY_LIMIT && x != 0;
4        @ ensures (\result ==> withdraw == \old(withdraw) + x) &&
5              (\old(withdraw) + x < DAILY_LIMIT ==> !\result) &&
6              \result ==> balance == \old(balance) + x ;
7        @ assignable balance;
8        @*/
9      boolean update(int x){...}
10     ...
11 }
```

Quelltext 3.8: Komposition mit Conjunctive Contract Refinement

die Callees schwer zu erfüllen, da die Callees alle Kontrakte kennen müssen und alle Nachbedingungen erfüllen müssen. Wie auch Conjuctive Contract Refinement ist Cumulative Contract Refinement durch die Eigenschaften von Disjunktion und Konjunktion assoziativ, idempotent und kommutativ [Thüm, 2015].

Da Cumulative Contract Refinement caller-preserving ist, muss für den komponierten Frame gelten $\alpha'' \subseteq \alpha$ und $\alpha'' \subseteq \alpha'$, damit erhalten wir $\alpha'' \subseteq \alpha' \cap \alpha$. Zusätzlich können wir wieder die gleiche Überlegung wie im Conjunctive Contract Refinement nutzen, dass der Frame auch mit der Nachbedingung ausgedrückt werden kann (siehe Abschnitt 3.1.5), weshalb wir den Frame Cut verwenden können:

$$c' \bullet_{CumCR} c = \{\phi' \vee \phi\}\, m' \bullet m\, \{\psi' \wedge \psi\}\, [\alpha' \cap \alpha]$$

Wegen der vorangestellten Herleitung bleibt Cumulative Contract Refinement caller-preserving. Assoziativität, Idempotenz und Kommutativität werden wegen der Eigenschaften von Schnittmengen erhalten.

Beispiel 3.6. Bei Cumulative Contract Refinement sieht der Kontrakt für `update` in *DailyLimit* so aus, wie bei Conjunctive Contract Refinement in Quelltext 3.7. Quelltext 3.9 zeigt die Komposition der Kontrakte, wobei wir wieder dieselben Probleme im Frame haben wie in Conjunctive Contract Refinement. Außerdem ist hier die Verknüpfung der Vorbedingung mit Disjunktion eher ungünstig, da wir eigentlich wollen, dass beide Vorbedingungen erfüllt sind, und hier nur eine von beiden erfüllt werden muss.

3.1.7 Consecutive Contract Refinement

Das *Consecutive Contract Refinement* [Thüm, 2015] benutzt Specification Inheritance nach Dhara und Leavens [Dhara and Leavens, 1996]. *Specification Inheritance* lockert die Regel, die in Cumulative Contract Refinement genutzt wird, indem eine Nachbedingung nur erfüllt werden muss, wenn die dazugehörige Vorbedingung erfüllt ist, dadurch müssen die Callees nicht mehr alle Nachbedingungen erfüllen. Consecutive Contract Refinement wird deshalb wie folgt ausgedrückt:

$$c' \bullet_{ConsCR} c = \{\phi' \vee \phi\}\, m' \bullet m\, \{(old\,(\phi') \Rightarrow \psi') \wedge (old\,(\phi) \Rightarrow \psi)\}$$

http://dx.doi.org/10.24355/dbbs.084-201711280920

```
1  public final class Account {                                    feature module CumCR
2      ...
3      /*@ requires withdraw >= DAILY_LIMIT || x != 0;
4        @ ensures (\result ==> withdraw == \old(withdraw) + x) &&
5              (\old(withdraw) + x < DAILY_LIMIT ==> !\result) &&
6              \result ==> balance == \old(balance) + x ;
7        @ assignable balance;
8        @*/
9      boolean update(int x){...}
10     ...
11 }
```

Quelltext 3.9: Komposition mit Cumulative Contract Refinement

Consecutive Contract Refinement verhält sich wie Cumulative Contract Refinement, bis auf das Problem, dass die Callees nicht mehr alle Nachbedingungen erfüllen müssen [Thüm, 2015].

Für den Frame wäre es plausibel ebenso zu verfahren, sodass der Frame von der Vorbedingung abhängt. Genau genommen wollen wir damit den Frame Cut etwas lockern. Das kann wieder mit dem Frame als Nachbedingung begründet werden. Wenn eine Vorbedingung erfüllt ist, werden also die entsprechenden nicht geänderten Felder zur Nachbedingung hinzugefügt. Dementsprechend erhalten wir einen Extended Frame Cut und das Consecutive Contract Refinement sieht wie folgt aus (x bezeichnet hier ein Feld aus dem Frame):

$$c' \bullet_{ConsCR} c = \{\phi' \vee \phi\}\, m' \bullet m\, \{(old\,(\phi') \Rightarrow \psi') \wedge (old\,(\phi) \Rightarrow \psi)\} \\ [\{x|\,(x \in \alpha' \wedge \phi')\} \cap \{x|\,(x \in \alpha \wedge \phi)\}]$$

Da wir den Frame analog zur Nachbedingung aufbauen, lassen sich die Beweise für Assoziativität, Idempotenz und Kommutativität analog zu den Beweisen der Nachbedingung [Thüm, 2015] ausführen. Die Caller-Kompatibilität lässt sich dementsprechend ebenfalls analog zur Nachbedingung aufbauen. Allerdings können wir den Frame so nicht in JML ausdrücken, da hier nur Felderlisten beziehungsweise Datengruppen gestattet sind [Leavens et al., 2008].

Vom Konzept her entspricht das Consecutive Contract Refinement für Vor- und Nachbedingungen den Spezifikationsfällen, die als Sprachkonzept in JML vorhanden sind (siehe Abschnitt 2.2). Dabei wird die obige Formel für Vor- und Nachbedingung vereinfacht, indem für jede Methodenverfeinerung ein Spezifikationsfall mit `normal_behavior` angelegt wird. Die Spezifikationsfälle werden anschließend mit `also` verknüpft. Für jeden Spezifikationsfall kann auch ein Frame angegeben werden, was ebenfalls eine Option für Consecutive Contract Refinement wäre. Das Problem, was hierbei aber besteht ist, dass die Caller-Kompatibilität nicht mehr gegeben ist. Das lässt sich einfach begründen, indem beide Vorbedingungen erfüllt sind und damit beide Frames zur Kontraktkomposition hinzugefügt werden, ist der komponierte Frame weder Teilmenge des Originalframes noch der Frameverfeinerung.

http://dx.doi.org/10.24355/dbbs.084-201711280920

Beispiel 3.7. Für Consecutive Contract Refinement zeigen wir die Darstellung mit Spezifikationsfällen in JML in Quelltext 3.11. Dafür verwenden wir die JML Schlüsselwörter `normal_behavior` und `also`. Bei der Spezifikation in Quelltext 3.10 müssen wir auf keine Besonderheiten achten, wir geben einfach die benötigten Vor- und Nachbedingungen an, sowie den benötigten Frame.

feature module *DailyLimit*

```
1  public final class Account {
2      ...
3      /*@ requires withdraw >= DAILY_LIMIT;
4        @ ensures (\result ==> withdraw == \old(withdraw) + x) &&
5              (\old(withdraw) + x < DAILY_LIMIT ==> !\result);
6        @ assignable withdraw;
7        @*/
8      boolean update(int x){...}
9      ...
10 }
```

Quelltext 3.10: Kontrakt für die Methode `update` im Feature *DailyLimit* für Spezifikationsfälle

composition *Specification Cases*

```
1  class Account {
2      ...
3      /*@ normal_behavior
4        @ requires x != 0;
5        @ ensures \result ==> balance == \old(balance) + x ;
6        @ assignable balance;
7        @ also
8        @ normal_behavior
9        @ requires withdraw >= DAILY_LIMIT;
10       @ ensures \result ==> withdraw == \old(withdraw) + x) &&
11             (\old(withdraw) + x < DAILY_LIMIT ==> !\result ;
12       @ assignable withdraw;
13       @*/
14     boolean update(int x){...}
15     ...
16 }
```

Quelltext 3.11: Komposition mit Spezifikationsfällen

3.1.8 Überblick zu den Kontraktkompositionsmechanismen

Wir haben sechs verschiedene Kontraktkompositionsmechanismen betrachtet und dafür Framing-Techniken anhand der Eigenschaften der Kontraktkompositionsmechanismen abgeleitet, die in Tabelle 3.2 mit ihren Eigenschaften dargestellt werden. Für Plain Contracting wird der Frame beibehalten, was wir im Folgenden als *Plain Framing* bezeichnen. Contract Overriding bildet das Komplement zum Plain Contracting und überschreibt den Frame mit dem *Frame Overriding*. Als nächstes haben wir für das Explicit Contract Refinement das Schlüsselwort `original` für den Frame erweitert, was wir nachfolgend als *Explicit Frame Refinement* bezeichnen. Anschließend haben wir für Conjunctive und Cumulative Contract Refinement den *Frame Cut*, der die Schnittmenge aus dem Originalframe und der Frameverfeinerung bildet.

http://dx.doi.org/10.24355/dbbs.084-201711280920

Kontraktkomposition		assoziativ	idempotent	kommutativ
$c' \bullet_{PC} c = \{\phi\}\, m' \bullet m\, \{\psi\}\, [\alpha]$	original-preserving	×	×	
$c' \bullet_{CO} c = \{\phi'\}\, m' \bullet m\, \{\psi'\}\, [\alpha']$	refinement-preserving	×	×	
$c' \bullet_{ECR} c = \{\phi' \bullet \phi\}\, m' \bullet m\, \{\psi' \bullet \psi\}\, [\alpha' \bullet \alpha]$	-	×		
$c' \bullet_{ConjCR} c = \{\phi' \wedge \phi\}\, m' \bullet m\, \{\psi' \wedge \psi\}\, [\alpha' \cup \alpha]$	-	×	×	×
$c' \bullet_{CumCR} c = \{\phi' \vee \phi\}\, m' \bullet m\, \{\psi' \wedge \psi\}\, [\alpha' \cup \alpha]$	caller-preserving	×	×	×
$c' \bullet_{ConsCR} c = \{\phi' \vee \phi\}\, m' \bullet m\, \{\psi''\}\, [\alpha'']$	caller-preserving[1]	×	×	×

mit $\psi'' = (old\,(\phi') \Rightarrow \psi') \wedge (old\,(\phi) \Rightarrow \psi)$
und $\alpha'' = \{x|\,(x \in \alpha' \wedge \phi')\} \cap \{x|\,(x \in \alpha \wedge \phi)\}$ für den Extended Frame Cut
oder $\alpha'' = \{x|\,(x \in \alpha' \wedge \phi') \vee (x \in \alpha \wedge \phi)\}$ für die Spezifikationsfälle

Tabelle 3.2: Übersicht der Kontraktkompositionsmechanismen

Zuletzt haben wir das Consecutive Contract Refinement mit dem *Extended Frame Cut*, als Erweiterung des Frame Cuts. Aber da der Extended Frame Cut sich nicht mit JML umsetzen lässt, haben wir zusätzlich noch die *Spezifikationsfälle* für das Consecutive Contract Refinement betrachtet.

Die Verwendung vom Frame Cut stellt allerdings ein Problem dar. Wie in den Beispielen zu Conjunctive und Cumulative Contract Refinement zu sehen war, ist die Technik zu restriktiv, da nur die Felder geändert werden dürfen, die in allen Kontraktverfeinerungen angegeben worden sind. Das stellt für die Feature-orientierte Programmierung eine Schwierigkeit dar, denn die Felder können durchaus erst in einem späteren Feature deklariert werden, wodurch es mit dem Schnitt allerdings nicht möglich wäre, diese Felder in den Frame aufzunehmen. Dementsprechend können wir den Frame Cut in der Form nicht für das Feature-orientierte Framing gebrauchen. Das bedeutet jedoch auch, dass wir für Conjunctive und Cumulative Contract Refinement eine bessere Framing-Technik brauchen.

Aber auch die anderen Framing-Techniken weisen Schwächen auf. Für Plain Framing haben wir bereits festgestellt, dass der Frame die Implementierungsmöglichkeiten beschränkt, da keine neuen Felder geändert werden dürfen. Beim Frame Overriding haben wir die erwähnte Problematik mit Inkonsistenzen durch Kopien in der Spezifikation. Explicit Contract Refinement enthält gegebenenfalls tote Referenzen und bietet keine Preserving-Eigenschaften. Die Spezifikationsfälle im Frame sorgen dafür, dass Consecutive Contract Refinement nicht mehr caller-preserving ist.

Da ein einziger Kontraktkompositionsmechanismus nicht ausreicht, um eine Spezifikation für alle Produktlinien zu erstellen [Thüm, 2015], ist es sinnvoll für alle Mechanismen passende Framing-Techniken zu haben. Eine Framing-Technik würde voraussichtlich genauso wenig reichen, wie ein Kontraktkompositionsmechanismus,

[1] Nur für den Extended Frame Cut, nicht für die Spezifikationsfälle

http://dx.doi.org/10.24355/dbbs.084-201711280920

da auch die Framing-Techniken unterschiedliche Schwächen aufweisen. Dementsprechend müssen wir noch betrachten, wie die Framing-Techniken aus diesem Abschnitt verbessert werden können, damit die Anwendbarkeit der Kontraktkompositionsmechanismen durch ihre Framing-Techniken nicht eingegrenzt werden.

3.2 Anwendung von Objekt-orientiertem Framing auf Feature-orientierte Programmierung

In diesem Abschnitt untersuchen wir existierende Framing-Techniken für Methodenüberschreibung, die in der Objekt-Orientierung (Vererbung) angewendet werden, damit wir daraus eine Strategie für die Kontraktkomposition im Feature-orientierten Framing entwickeln können. Dazu betrachten wir Framing-Techniken, die sich mit Framing bei Vererbung befassen, da die Probleme bei der Vererbung denen in der Feature-orientierten Programmierung ähneln. Anschließend analysieren wir, wie sich die Framing-Techniken auf Feature-orientiertes Framing anwenden lassen.

Zunächst begutachten wir die Gemeinsamkeiten von Feature-Orientierter Programmierung und Vererbung. Die Gemeinsamkeit stellt sich insbesondere durch die Konstrukte `super` [Ullenboom, 2004] und `original` heraus. In der Objekt-orientierten Vererbung wird mit `super` auf die Methode der Superklasse zugegriffen und so erweitert oder überschrieben [Ullenboom, 2004]. Das Äquivalent für die Feature-orientierte Programmierung `original` kann auf die Methode des vorherigen Features zugreifen [Apel et al., 2013b]. Zudem können sowohl in Subklassen als auch in Features neue Felder zu den Klassen hinzugefügt werden [Ullenboom, 2004, Apel et al., 2013b]. Deshalb lohnt es sich, Konzepte für Framing in Vererbung hinsichtlich ihrer Relevanz für Feature-orientiertes Framing zu analysieren.

Die Unterschiede von Feature-orientierter Programmierung und Vererbung müssen auch berücksichtigt werden. Vom Konzept her bieten die Subklassen eine Spezialisierung der Superklasse an [Ullenboom, 2004], während Features eine gekapselte Funktionalität darstellen [Prehofer, 1997]. Außerdem sind die Features optional [Prehofer, 1997], die Subklassen aber nicht [Ullenboom, 2004].

Beispiel 3.8. Sehen wir uns dazu die Methode `update` in Quelltext 2.1 und Quelltext 2.2 an. Um daraus eine Objekt-orientierte Vererbung zu erhalten, müssen wir die Deklarationen der beiden Klassen ändern und in der Klasse im Feature *DailyLimit* das Schlüsselwort `original` durch das Schlüsselwort der Objekt-Orientierung `super` ersetzen. Bei der Deklaration der Klasse `Account` im Feature *BankAccount* muss das `final` entfernt werden, damit die Klasse verfeinert werden darf. Im Feature *DailyLimit* müssen wir die Klasse umbenennen und mit dem Schlüsselwort `extends` angeben, dass die Klasse `Account` erweitert. Mit den Änderungen ergibt sich für die Subklasse Quelltext 3.12.

3.2.1 Behavioral Subtyping

Ein Problem von Framing bei Vererbung ist, dass eine Subklasse den Kontrakt der Superklasse erfüllen muss [Liskov and Wing, 1994, Hatcliff et al., 2012], damit die

http://dx.doi.org/10.24355/dbbs.084-201711280920

```
1  class AccountDailyLimit extends Account {
2      public final static int DAILY_LIMIT = -1000;
3      public int withdraw = 0;
4
5      boolean update(int x) {
6          int newWithdraw = withdraw;
7          if (x < 0)  {
8              newWithdraw += x;
9              if (checkWithdraw(x, false))
10                 return false;
11         }
12         if (!super.update(x))
13             return false;
14         withdraw = newWithdraw;
15         return true;
16     }
17 }
```

Quelltext 3.12: Feature *DailyLimit* als Subklasse

Subklasse wie die Superklasse verwendet werden kann. Wenn die Subklasse die Vorbedingung verstärkt, können nicht alle Caller, die die Superklasse aufrufen, die Subklasse wie die Superklasse verwenden, da der Caller nicht alle Vorbedingungen für den Aufruf der Subklasse erfüllt. Ähnliches gilt für das Abschwächen der Nachbedingung, dann kann sich der Caller nicht auf die Nachbedingungen der Superklasse verlassen, da die Subklasse diese womöglich nicht erfüllt. *Behavioral Subtyping* legt Bedingungen fest, wodurch der Kontrakt der Subklasse den Kontrakt der Superklasse erfüllt. Detailliert heißt das für eine Superklasse C mit Methode m und deren Verfeinerung m' in Subklasse D [Hatcliff et al., 2012]:

- Für eine Methode $D.m'$, die eine Methode $C.m$ überschreibt, gilt, dass die Vorbedingung von $D.m'$ die Vorbedingung von $C.m$ impliziert, das heißt die Vorbedingung kann abgeschwächt werden.
- Für eine Methode $D.m'$, die eine Methode $C.m$ überschreibt, gilt, dass die Nachbedingung von $C.m$ die Nachbedingung von $D.m'$ impliziert, das heißt die Nachbedingung kann verstärkt werden.
- Für eine Methode $D.m'$, die eine Methode $C.m$ überschreibt, gilt, dass die Menge der modifizierbaren Variablen von $D.m'$ eine Teilmenge der modifizierbaren Variablen von $C.m$ sind.

Durch diese Bedingungen sind die Möglichkeiten für die Implementierung eingeschränkt, insbesondere dadurch, dass wir dem Frame keine neu definierten Felder hinzufügen dürfen, wie das folgende Beispiel zeigt.

Beispiel 3.9. Betrachten wir als Beispiel für die Einschränkung eine Superklasse `Account` (Quelltext 3.13), die ein Bankkonto repräsentiert, und ihre Subklasse `AccountDailyLimit` (Quelltext 3.14), die ein tägliches Abhebungslimit zum Bankkonto hinzufügt. Dafür muss die Methode `update` zusätzlich zum Feld `balance` auch noch das Feld `withdraw` modifizieren, wie in Zeile 12 zu sehen ist. Nach Behavioral

http://dx.doi.org/10.24355/dbbs.084-201711280920

Subtyping wäre dies nicht möglich, da so der Kontrakt von `Account.update` in Zeile 6 verletzt werden würde. Somit wäre die Implementierung in Quelltext 3.14 als Subklasse nicht möglich.

```
1  public class Account {
2      /*@
3        @ requires x != 0;
4        @ ensures \result ==> \old(balance) + x >= OVERDRAFT_LIMIT;
5        @ ensures \old(balance) + x < OVERDRAFT_LIMIT ==> !\result;
6        @ assignable balance;
7        @*/
8      boolean update(int x) {...}
9
10 }
```

Quelltext 3.13: Superklasse `Account`

```
1  class AccountDailyLimit extends Account{
2      public final static int DAILY_LIMIT = -1000;
3      public int withdraw = 0;
4
5      /*@
6        @ requires x != 0;
7        @ ensures \result ==> (((x < 0) ==>
8        @          (\old(withdraw) + x >= DAILY_LIMIT))
9        @          || \old(balance) + x >= OVERDRAFT_LIMIT);
10       @ ensures (((x < 0) && (\old(withdraw) + x < DAILY_LIMIT))
11       @          || \old(balance) + x < OVERDRAFT_LIMIT)) ==> !\result;
12       @ assignable balance, withdraw;
13       @*/
14     boolean update(int x) {...}
15 }
```

Quelltext 3.14: Subklasse mit täglichem Abhebungslimit

Für Delta-orientierte Programmierung wird das Prinzip vom Behavioral Subtyping von Hähnle und Schaefer auf die Deltas übertragen [Hähnle and Schaefer, 2012]. Dazu wird das Behavioral Subtyping wie folgt angewandt, sodass für einen modifizierten Kontrakt gilt (mit Klasse C, Methode m, requires-Klauseln r und r', ensures-Klauseln e und e' und assignable-Klauseln a und a'):

1. Für die Vorbedingung: $C.m.r.original \vee r'$
2. Für die Nachbedingung: $C.m.e.original \wedge e'$
3. Für das Framing: $a' \subseteq C.a.original$

Wie wir bereits in dem Beispiel Quelltext 3.13 und Quelltext 3.14 für Vererbung festgestellt haben, schränkt Behavioral Subtyping die Möglichkeiten der Implementierung ein. Da wir dem Programm mit einem Feature Funktionalität hinzufügen wollen, ist die Einschränkung hinderlich, da so, wie auch in der Vererbung, nicht

http://dx.doi.org/10.24355/dbbs.084-201711280920

einmal die neu hinzugefügte Felder in verfeinerten Methoden geändert werden dürfen. Für Feature-orientierte Kontrakte eignet sich Behavioral Subtyping, wie es hier dargestellt ist, also nicht [Thüm, 2015], um Behavioral Subtyping zu verwenden, muss es möglich sein die Beschränkungen für den Frame zu umgehen. Eine Möglichkeit dafür betrachten wir im folgenden Abschnitt.

3.2.2 Datengruppen

Leino bietet mit den Datengruppen [Leino, 1998] eine Möglichkeit, um einen validen Frame für Subklassen zu Erzeugen. Eine *Datengruppe* wird in einer Klasse in Annotationen deklariert und repräsentiert eine Menge von Variablen, die in den Kontrakten verwendet werden können. Um eine Variable zu einer oder mehreren Datengruppen hinzuzufügen, muss die entsprechende Datengruppe bei der Deklaration des Felds in einer Annotation angegeben werden, die Felder dürfen also nicht mehr nachträglich einer Datengruppe zugeordnet werden. Datengruppen dürfen auch andere Datengruppen enthalten, das heißt sie können verschachtelt werden. Die Datengruppen können nun für das Framing verwendet werden, wodurch in Subklassen neu deklarierte Variablen über die Datengruppe zum Frame hinzugefügt werden dürfen. Dadurch kann das Behavioral Subtyping eingehalten werden, da nur die Datengruppe Teil des Frames ist, aber es können trotzdem in der Subklasse neu deklarierte Felder dem Frame hinzugefügt werden.

Die Datengruppen werden von JML unterstützt [Leavens et al., 2006]. Dafür wird für jedes deklarierte *model field* eine Datengruppe mit demselben Namen automatisch generiert, das heißt das model field kann parallel als Spezifikationsvariable und als Datengruppe verwendet werden. Der Datentyp des model fields ist für die Datengruppe aber nicht von Belang. Jedes model field gehört automatisch zu der Datengruppe mit dem gleichen Namen. Weitere Felder können entweder mit der *in-Klausel* oder der *maps-into-Klausel* zu der Datengruppe hinzugefügt werden, welche direkt nach der Deklaration des Feldes erfolgen muss. Die in-Klausel fügt das Feld der Datengruppe hinzu, während mit der maps-into-Klausel die Felder eines Objekts hinzugefügt werden, zum Beispiel die Felder eines Arrays. Die model fields können dann in der assignable-Klausel verwendet werden, um den Frame zu definieren.

Beispiel 3.10. Betrachten wir hierfür wieder das Beispiel (Quelltext 3.13 und Quelltext 3.14) mit der Klasse `Account` und ihrer Subklasse `AccountDailyLimit`, bei denen wir für ein tägliches Abhebungslimit in `AccountDailyLimit` das Feld `withdraw` modifizieren müssen. Dafür können wir nun in der Klasse `Account` eine Datengruppe `updateDataGroup` erzeugen, der noch mit der in-Klausel das Feld `balance` hinzugefügt wird in Quelltext 3.15. Um den Kontrakt der Superklasse einzuhalten und trotzdem `withdraw` modifizieren zu können, muss jetzt nur noch in `AccountDailyLimit` in Quelltext 3.16 `withdraw` der Datengruppe `updateDataGroup` hinzugefügt werden.

Um die Datengruppen auf Feature-orientiertes Framing anzuwenden, müssen wir überlegen, wie sich das Prinzip von Vererbung auf Feature-orientierte Programmierung übertragen lässt. Dafür müssen wir wieder in einer beliebigen Klasse die Datengruppe definieren können. Anschließend dürfen die Felder bei ihrer Deklaration einer

http://dx.doi.org/10.24355/dbbs.084-201711280920

```
1 public class Account {
2     //@ public model instance int updateDataGroup;
3     public int balance = 0; //@ in updateDataGroup;
4
5     //@ assignable updateDataGroup;
6     boolean update(int x){...}
7     ...
8 }
```

Quelltext 3.15: Superklasse `Account` mit Datengruppe

```
1 class AccountDailyLimit extends Account{
2     public int withdraw = 0; //@ in updateDataGroup;
3     ...
4 }
```

Quelltext 3.16: Subklasse mit täglichem Abhebungslimit mit Datengruppe

oder mehreren Datengruppen hinzugefügt werden. Zusätzlich muss bei der Komposition der Features überlegt werden, wie der Frame zusammengefügt werden soll. Eine sinnvolle Möglichkeit wäre es, Plain Framing in Kombination mit einer oder mehreren Datengruppen zu verwenden, die im ersten Feature deklariert werden und in allen weiteren Features werden nur Felder zu den Datengruppen hinzugefügt. Alternativ können wir die Datengruppen in allen Features im Frame angeben, sodass überall indirekt derselbe Frame angegeben ist, und den Frame Cut verwenden.

Wenn wir die JML Notation für Datengruppen für Feature-orientiertes Framing verwenden wollen, müssen wir ein paar Dinge beachten. Für die Definition der Datengruppen können einfach die model fields verwendet werden und für die Zuordnung zu einer Datengruppe die in-Klausel oder die maps-into-Klausel. Ebenso können die Datengruppen in der assignable-Klausel verwendet werden. Die größte Herausforderung stellt die Komposition der Features dar. Insbesondere können Felder in Feature-orientierter Programmierung doppelt eingeführt werden, zum Beispiel um den initialen Wert des Feldes zu ändern. Also müssen wir hier als Beschränkung festlegen, dass nur bei der ersten Deklaration eines Feldes die Datengruppen angegeben werden können. Als nächstes sollten nur über die Datengruppen Felder zur assignable-Klausel indirekt hinzugefügt werden, damit wir die Vorteile der Datengruppen überhaupt nutzen können. Das können wir mit der oben vorgeschlagenen Verwendung von Plain Contracting und Frame Cut erreichen. Zusätzlich muss bei der Komposition der Features die Annotationen der Felder, also die in-Klauseln und die maps-into-Klauseln, übernommen werden.

Beispiel 3.11. In Quelltext 3.17 und Quelltext 3.18 sehen wir die Klasse `Account` für die Features *BankAccount* und *DailyLimit* mit Datengruppen in JML. Dazu wird im Feature *BankAccount* das model field `updateDataGroup` deklariert und damit die gleichnamige Datengruppe (Zeile 2). Zusätzlich fügen wir mit der in-Klausel `balance` im Feature *BankAccount* (Zeile 5) und im Feature *DailyLimit* `withdraw` zur Datengruppe `updateDataGroup` hinzu (Zeile 7).

http://dx.doi.org/10.24355/dbbs.084-201711280920

feature module *BankAccount*

```
1  public final class Account {
2      //@ public model instance int updateDataGroup;
3      ...
4      public int balance = 0; //@ in updateDataGroup;
5
6      /*@
7        @ requires ...
8        @ ensures ...
9        @ assignable updateDataGroup;
10       @*/
11     boolean update(int x) {...}
12     ...
13 }
```

Quelltext 3.17: Klasse `Account` im Feature *BankAccount* mit JML Datengruppen

feature module *DailyLimit*

```
1  class Account{
2      public final static int DAILY_LIMIT = -1000;
3      public int withdraw = 0; //@ in updateDataGroup;
4
5      /*@
6        @ requires ...
7        @ ensures ...
8        @ assignable updateDataGroup;
9        @*/
10     boolean update(int x) {...}
11     ...
12 }
```

Quelltext 3.18: Klasse `Account` im Feature *DailyLimit* mit JML Datengruppen

Der Vorteil von den Datengruppen gegenüber Behavioral Subtyping ist, dass wir den Kontrakt der Superklasse erfüllen, aber dennoch Felder dem Frame hinzufügen können, wodurch wir bei der Anwendung auf Feature-orientiertes Framing Preserving-Eigenschaften erhalten können. Welche Preserving-Eigenschaften wir erhalten ist abhängig davon, ob wir Plain Framing oder Frame Cut verwenden. Datengruppen in Kombination mit Plain Framing bewahren das original-preserving, da wir die Datengruppen aus dem Originalframe verwenden und damit $\alpha'' = \alpha$ gilt, während weder refinement-caller-preserving noch refinement-callee-preserving erhalten bleibt, da der verfeinerte Frame auch Felder oder Datengruppen enthalten kann, die nur bei der Komposition nicht berücksichtigt werden, und dadurch weder $\alpha'' \subseteq \alpha'$ für das refinement-caller-preserving noch $\alpha'' \supseteq \alpha'$ für das refinement-callee-preserving gelten muss. Wenn wir die Datengruppen wie oben beschrieben mit Frame Cut komponieren erhalten wir alle Preserving-Eigenschaften, da wir den Frame wiederholen und damit $\alpha'' = \alpha' = \alpha$ gilt, der Frame ändert sich nur indirekt durch das Hinzufügen von Feldern zu den Datengruppen. Allerdings müssen wir dafür in jedem Feature den Frame einer Methode wiederholen, wodurch hier mehr Spezifikationsaufwand als in der Technik mit Plain Framing entsteht. Damit können wir dafür das Problem umgehen, das wir im Frame Cut bei Conjunctive und Cumulative Contract Refinement hatten, durch das wir keine neu definierten Felder in den Frame aufnehmen konnten.

http://dx.doi.org/10.24355/dbbs.084-201711280920

Wir können außerdem angeben, dass Datengruppen idempotent, assoziativ und kommutativ sind. Dies kann einfach begründet werden, wenn ein Feature mit sich selbst komponiert wird, werden die Felder, wegen der Beschränkung für doppelte Einführung von Feldern, einmal der Datengruppe hinzugefügt, weshalb der Frame äquivalent zum Frame des Originalkontrakts ist, dementsprechend sind Datengruppen idempotent. Bei der Komposition von drei Features ist die Reihenfolge der Ausführung ebenfalls irrelevant, da die Felder unabhängig von der Reihenfolge den Datengruppen hinzugefügt werden und sowohl Plain Framing als auch Frame Cut assoziativ ist, sind Datengruppen assoziativ. Datengruppen mit Plain Framing sind nicht kommutativ, da Plain Framing nicht kommutativ ist, und Datengruppen mit Frame Cut sind kommutativ, da hier der Frame im Originalframe und im verfeinerten Frame gleich sind und die Reihenfolge damit keine Relevanz hat und Frame Cut selbst auch kommutativ ist.

Eine für die Datengruppe relevante Problematik ist das Abstract Aliasing [Leino and Nelson, 2002, Weiß, 2011]. *Abstract Aliasing* [Leino and Nelson, 2002] beschreibt das Auftreten von unerwarteten Nebeneffekten zwischen einer abstrakten Variable, die keinen zugeordneten Wert, sondern nur eine Repräsentation besitzen, (auch bekannt als model oder ghost Variablen) und ihrer Repräsentation. Die Nebeneffekte treten auf, wenn die Repräsentation eine Abhängigkeit zu einer Variable hat, die der abstrakten Variable nicht bekannt ist. Für das Framing heißt das, dass nicht sichergestellt werden kann, ob ein Feld zu dem Frame gehört oder nicht.

Mit Datengruppen in JML wird die Abwesenheit von Abstract Aliasing nicht sichergestellt [Weiß, 2011]. Die Abhängigkeiten können wir in statische Abhängigkeiten und dynamische Abhängigkeiten unterteilen. Die *statische Abhängigkeit* beschreibt eine Abhängigkeit zwischen einem model field und einem Feld des gleichen Objekts (`this`) und die *dynamische Abhängigkeit* ist eine Abhängigkeit zwischen einem model field und einem Feld, das nicht zum gleichen Objekt gehört, aber von einem Feld des gleichen Objekts referenziert wird. Die statischen Abhängigkeiten erhalten wir durch die in-Klausel, die dynamischen Abhängigkeiten durch die maps-into-Klauseln [Weiß, 2011]. Da die dynamischen Abhängigkeiten Felder eines anderen Objekts referenzieren, wissen wir nicht, ob das Feld weitere Abhängigkeiten enthält, wodurch unbekannt ist, ob ein beliebiges anderes Feld indirekt zum Frame gehört.

Beispiel 3.12. Wenn wir eine Liste `List<int> updates` zur Klasse `Account` in Quelltext 3.15 zur Datengruppe `updateDataGroup` hinzufügen, können wir bei der Ausführung von `update(1500)` in Quelltext 3.19 nicht bestimmen, ob `x` in `updateDataGroup` ist. Betrachten wir jetzt die Nachbedingung für `doSomething` $x == x+1$, können wir die Nachbedingung nicht beweisen, da `update` das Feld `x` modifiziert haben könnte.

Ebenfalls problematisch für die Verifikation mit Datengruppen ist das Local Reasoning für eine Klasse [Ahrendt et al., 2016, Weiß, 2011]. Das bedeutet, dass wir lokal in einer Klasse nicht feststellen können, welchen Inhalt eine Datengruppe hat. Das liegt daran, dass die Felder nach ihrer Deklaration hinzugefügt werden. Da auch Felder aus anderen Klassen den Datengruppen hinzugefügt werden dürfen, können wir

http://dx.doi.org/10.24355/dbbs.084-201711280920

```
1  public class Main {
2      public static int x;
3
4      /*@ normal_behavior
5        @ ensures x == \old(x)+1;
6        @*/
7      void doSomething(Account a){
8          a.update(1500);
9          x++;
10     }
11 }
```

Quelltext 3.19: `Main` Klasse mit Abstract Aliasing

lokal in der Klasse der Datengruppe nicht feststellen, welche Felder aus den anderen Klassen sich in der Datengruppe befinden. Das ist insbesondere dann kritisch, wenn wir einen Kontrakt mit einer Datengruppe beweisen wollen und ein Feld aus einer anderen Klasse geändert wird, da dann bewiesen werden muss, dass das Feld in der Datengruppe ist.

Ein weiterer Nachteil ist der Spezifikationsaufwand. Um Datengruppen zu verwenden, müssen wir die Annotationen in jeder Klasse bei jedem Feld hinzufügen und für alle Methoden Datengruppen für die Frames definieren, was im Zweifelsfall eine Datengruppe pro Methode ist[2]. Bei größeren Programmen ist der Aufwand für das Einführen der Datengruppen damit nicht unerheblich, dafür erleichtern wir das Framing. Hier muss also zwischen dem Nutzen und den Kosten für die Verwendung von Datengruppen abgewogen werden.

3.2.3 Dynamic Frames

Kassios [Kassios, 2006] hat als Lösung für Abstract Aliasing im Kontext von Framing Dynamic Frames eingeführt. *Dynamic Frames* sind spezielle Spezifikationsvariablen beziehungsweise model fields oder pure methods, die als Mengen verwendet werden, deren Inhalt sich während der Programmausführung ändern kann, ähnlich wie bei den Datengruppen. Allerdings werden zusätzlich die elementaren Mengenoperationen für die Dynamic Frames definiert, sodass zum Beispiel angegeben werden kann, dass spezielle Variablen nicht Teil des Frames sind oder der Frame die Überlappung von zwei Mengen ist.

JML* von Weiß [Weiß, 2011] ist eine Erweiterung von JML, die Dynamic Frames ermöglicht. Dafür wird zunächst ein Dynamic Frame erstellt, der durch ein model field vom Typ `\locset` repräsentiert wird. Der Inhalt des Dynamic Frames kann anschließend mit der represents-Klausel angegeben werden, welche einmal pro Klasse und Feld angewendet werden kann. In der represents-Klausel muss letztlich der gesamte Frame angegeben werden und nicht nur die Teile des Frames, die hinzugefügt werden sollen, das heißt der Nutzer muss wissen, welche Felder aus der Superklasse

[2] Wenn wir Spezifikationsfälle verwenden könnten auch mehrere Datengruppen für eine Methode benötigt werden.

http://dx.doi.org/10.24355/dbbs.084-201711280920

im Frame stehen müssen. Einerseits wird die Verwendung gegenüber Datengruppen so schwieriger, dafür ist aber Local Reasoning möglich. Die Mengenoperationen können mit `\intersect`, `\set_minus`, `\set_union`, `\subset` und `\disjoint` abgerufen werden.

Beispiel 3.13. Wir betrachten das Beispiel der Datengruppe jetzt mit Dynamic Frames. In Quelltext 3.20 deklarieren wir den Dynamic Frame `updateFrame`, welcher als Frame für die Methode `update` genutzt wird. In *BankAccount* enthält der Frame `balance` und in Quelltext 3.21 wird er auf `withdraw` erweitert, aber das Feld `balance` muss ebenfalls aufgeführt werden, da die represents-Klausel nur überschrieben, aber nicht erweitert, werden kann. Die Problematik, die wir in Quelltext 3.19 hatten, wird in Quelltext 3.22 gelöst, indem wir als Vorbedingung aufführen, dass das Feld `x` nicht in dem Frame von `update` liegt, wodurch wir die Nachbedingung von `doSomething` jetzt beweisen können.

feature module *BankAccount*

```
1  public class Account {
2      //@ public model \locset updateFrame;
3      public int balance = 0;
4      //@ public represents updateFrame = balance;
5
6      /*@
7        @ requires x != 0;
8        @ ensures \result ==> \old(balance) + x >= OVERDRAFT_LIMIT;
9        @ ensures \old(balance) + x < OVERDRAFT_LIMIT ==> !\result;
10       @ assignable updateFrame;
11       @*/
12     boolean update(int x) {...}
13     ...
14 }
```

Quelltext 3.20: Klasse `Account` im Feature *BankAccount* mit Dynamic Frame

feature module *DailyLimit*

```
1  class DailyLimitAccount extends Account{
2      public final static int DAILY_LIMIT = -1000;
3      public int withdraw = 0;
4      //@ public represents updateFrame = balance,withdraw;
5      /*@
6        @ requires x != 0;
7        @ ensures \result ==> (((x < 0) ==>
8        @                  (\old(withdraw) + x >= DAILY_LIMIT))
9        @              || \old(balance) + x >= OVERDRAFT_LIMIT);
10       @ ensures (((x < 0) && (\old(withdraw) + x < DAILY_LIMIT))
11       @      || \old(balance) + x < OVERDRAFT_LIMIT)) ==> !\result;
12       @ assignable updateFrame;
13       @*/
14     boolean update(int x) {...}
15     ...
16 }
```

Quelltext 3.21: Klasse `Account` im Feature *DailyLimit* mit Dynamic Frame

http://dx.doi.org/10.24355/dbbs.084-201711280920

```
1  public class Main {
2      public int x;
3
4    /*@ normal_behavior
5      @ requires \disjoint(a.updateFrame, this.x);
6      @ ensures x == \old(x)+1;
7      @*/
8      void doSomething(Account a){
9          a.update(1500);
10         x++;
11     }
12 }
```

Quelltext 3.22: `Main` Klasse ohne Abstract Aliasing

Um das Prinzip von Dynamic Frames auf Feature-orientierte Programmierung anzuwenden, müssen wir überlegen, wie die Komposition von den Klassen letztlich aussehen soll. Gehen wir von JML* aus, hat jedes Feature gegebenenfalls eine represents-Klausel für einen Dynamic Frame. Da in jeder Klasse nur eine represents-Klausel pro model field verwendet werden darf, können nicht alle Klauseln eins zu eins übetragen werden, das heißt die einzige Option wäre es hier bei der Komposition von zwei Features die represents-Klauseln zu kombinieren. Dafür stehen uns zwei Optionen zur Verfügung. Entweder wir überschreiben die Klausel oder wir geben anstatt des gesamten Frames in der represents-Klausel eines Features nur den Teil, der ergänzt werden soll, an. So würde beispielsweise in Quelltext 3.21 statt [`represents updateFrame = balance,withdraw;`] [`represents updateFrame = withdraw;`] stehen. Außerdem müssen wir die Frames selbst auch wieder komponieren, was wir genau wie bei den Datengruppen mit Plain Framing und Frame Cut handhaben (siehe Abschnitt 3.2.2). Für Frame Cut muss wieder gelten, dass die Frames in den verschiedenen Features indirekt gleich sind.

Durch die represents-Klausel ist es möglich beliebige Felder zum Frame hinzuzufügen, was bei Datengruppen nicht möglich war, um die Caller-Kompatibilität der Subklasse zur Superklasse zu gewährleisten. Da wir aber insbesondere für Kompositionsmechanismen mit Caller-Kompatibilität neue Framing-Techniken brauchen, müssen wir auch hier eine Restriktion angeben. Also geben wir an, dass nur in einem Feature neu definierte Felder dem Frame hinzugefügt werden dürfen.

Jetzt stellt sich die Frage, welche Vorteile wir durch die Verwendung der Dynamic Frames haben. Wie schon bei den Datengruppen können wir abhängig von Plain Framing und Frame Cut entweder original-preserving oder alle Preserving-Eigenschaften im Frame durch die Verwendung von Dynamic Frames erhalten, da auch hier der Frame nur indirekt verändert wird. Im Gegensatz zu Datengruppen haben wir kein Abstract Aliasing, wegen der neuen Operatoren. Außerdem ist Local Reasoning gegeben, da in der represents-Klausel geprüft werden kann, welche Felder in dem Dynamic Frame enthalten sind.

http://dx.doi.org/10.24355/dbbs.084-201711280920

Dafür haben wir im Kontext von Feature-orientiertem Framing Nachteile zu verzeichnen. Wenn wir die represents-Klausel zum Überschreiben der vorangegangenen represents-Klausel nutzen, haben wir die aus dem Contract Overriding bekannte Problematik von Kopien. Wir müssen demnach in einer Klassenverfeinerung den Inhalt der represents-Klausel der Originalklasse kopieren. Dadurch entstehen bei Änderungen Inkonsistenzen und was insbesondere ein Problem darstellt ist, dass manche Features optional sind und so gegebenenfalls bei nicht gewähltem Feature Felder im Frame stehen, die nicht existieren. Dementsprechend können wir die für Superklassen und Subklassen gedachte Implementierung so nicht für Feature-orientiertes Framing verwenden. Verwenden wir die von uns alternativ vorgeschlagene Notation, dass in jedem Feature nur die neu hinzugefügten Felder in die represents-Klausel schreiben, besteht das Problem nicht oder wir können überlegen, wie in Explicit Contract Refinement, ein `original` für die represents-Klausel einzuführen, wodurch wir die represents-Klausel des vorherigen Features aufrufen können. Wie schon bei den Datengruppen, haben wir unabhängig von der gewählten Umsetzung einen gewissen manuellen Aufwand, da die Spezifikationen angepasst werden müssen. Wobei der Aufwand gegenüber den Datengruppen größer ist, da zusätzlich zu den represents-Klauseln und den model fields auch noch die Vorbedingungen geändert werden müssen, um gegebenenfalls die Abwesenheit des Abstract Aliasing zu sichern.

Da sich die Option mit dem Überschreiben der represents-Klausel nicht eignet, betrachten wir Idempotenz, Assoziativität und Kommutativität nur für die Optionen, dass nur die neu hinzugefügten Felder in der represents-Klausel angegeben werden oder wir `original` verwenden mit Frame Cut (da dieser alle Eigenschaften erfüllt). Für die erste Option haben wir festgelegt, dass die represents-Klauseln zusammengefügt werden, das heißt wenn zwei Klassen komponiert werden, erhalten wir eine Vereinigung der beiden Mengen aus den represents-Klauseln. Da die Vereinigung von Mengen, idempotent, assoziativ und kommutativ sind, folgen die Eigenschaften für die Dynamic Frames. Bei der Option mit `original` sieht es anders aus, denn wenn wir das `original` nicht verwenden, überschreiben wir den Frame, weshalb die Option nicht kommutativ ist. Aber analog zum `original` aus dem Explicit Contract Refinement können wir sagen, dass die Option assoziativ und idempotent ist. Mit Plain Framing bei Dynamic Frames erhalten wir analog zu den Datengruppen, dass die Technik nicht kommutativ ist unabhängig davon, ob `original` verwendet wird oder nicht.

3.3 Zusammenfassung

Wie wir festgestellt haben, gibt es viele Möglichkeiten, wie wir das Framing umsetzen können (siehe Tabelle 3.3). In Abschnitt 3.1 haben wir die Framing-Techniken Plain Framing, Frame Overriding, Explicit Frame Refinement, Frame Cut und Spezifikationsfälle vorgestellt. Zusätzlich haben wir die Techniken Behavioral Subtyping, Datengruppen und Dynamic Frames in Abschnitt 3.2 analysiert.

Datengruppen und Dynamic Frames bieten uns eine neue Möglichkeit für die Framekomposition an, die wir im Kontext der Kontraktkompositionsmechanismen verwenden wollen. Bei beiden hängen die Eigenschaften davon ab, ob wir Plain Framing

http://dx.doi.org/10.24355/dbbs.084-201711280920

Framing-Technik		assoziativ	idempotent	kommutativ
Plain Framing	original-preserving	×	×	
Frame Overriding	refinement-preserving	×	×	
Explicit Frame Refinement	-	×		
Frame Cut	caller-preserving	×	×	×
Consecutive Frame Refinement	-	×	×	×
Datengruppen mit Plain Framing	original-preserving	×	×	
Datengruppen mit Frame Cut	all-preserving	×	×	×
Dynamic Frames mit Plain Framing	original-preserving	×	×	
Dynamic Frames mit Frame Cut	all-preserving	×	×	×

all-preserving = refinement-preserving ∧ original-preserving

Tabelle 3.3: Übersicht über die Framing-Techniken

oder Frame Cut verwenden, wie in Tabelle 3.3 zu sehen ist. Wenn wir Datengruppen und Dynamic Frames jetzt für einen Kontraktkompositionsmechanismus verwenden wollen, müssen wir überlegen, welche Version wir verwenden. Für Plain Contracting und Explicit Contract Refinement reicht jeweils die Version mit Plain Framing, da wir nicht mehr als original-preserving garantieren und die Mechanismen nicht kommutativ sind und wir so weniger Spezifikationsaufwand haben. Dagegen brauchen wir für Contract Overriding und Cumulative Contract Refinement die Version mit Frame Cut, da hierfür zusätzlich refinement-caller-preserving bei beiden und refinement-callee-preserving bei Contract Overriding gelten muss. Ebenso benötigen wir die Version mit Frame Cut für Conjunctive und Consecutive Contract Refinement, da die Techniken kommutativ sind und Datengruppen/Dynamic Frames mit Plain Framing nicht.

Bei Contract Overriding und Explicit Contract Overriding haben wir auch die Möglichkeit, dass Vor- und Nachbedingungen entfernt werden können, was auch Konsequenzen für den Frame hat. Für Contract Overriding müssen wir die Vor- und Nachbedingung, die entfernt werden soll, einfach nicht wiederholen und bei Explicit Contract Refinement wiederholen wir die Vor- oder Nachbedingung nicht und nutzen kein `original`. Wenn wir eine Nachbedingung entfernen, werden eventuell Felder nicht mehr bei der Durchführung einer Methode geändert, also könnten sie auch aus dem Frame gelöscht werden. Da wir den Frame möglichst klein halten wollen, um die Verifikation zu erleichtern, ist es durchaus sinnvoll bei Contract Overriding und Explicit Contract Refinement Felder aus dem Frame löschen zu können. Dementsprechend würden wir weder Datengruppen noch Dynamic Frames für Contract Overriding und Explicit Contract Refinement verwenden, da beide Techniken kein Entfernen von Feldern aus dem Frame erlauben (zumindest nicht bei unseren ausgewählten Beschränkungen).

http://dx.doi.org/10.24355/dbbs.084-201711280920

Kompositionsmechanismus	Framing-Technik
PC	Plain Framing, Datengruppen/Dynamic Frames
CO	Frame Overriding
ECR	Explicit Frame Refinement
ConjCR	Frame Cut, Datengruppen/Dynamic Frames
CumCR	Frame Cut, Datengruppen/Dynamic Frames
ConsCR	Spezifikationsfälle, Datengruppen/Dynamic Frames

Tabelle 3.4: Kombinationen von Kontraktkompositionsmechanismen und Framing-Techniken

Damit erhalten wir die Kombinationen, die wir in Tabelle 3.4 angegeben haben. Wir können Plain Contracting mit Plain Framing mit und ohne Datengruppen und Dynamic Frames verwenden. Contract Overriding benutzt Frame Overriding. Explicit Contract Refinement verwendet das Explicit Frame Refinement. Conjunctive und Cumulative Contract Refinement können beide den Frame Cut mit oder ohne Datengruppen und Dynamic Frames verwenden. Consecutive Contract Refinement kann die Spezifikationsfälle verwenden oder Datengruppen und Dynamic Frames mit Frame Cut.

http://dx.doi.org/10.24355/dbbs.084-201711280920

4. Evaluierung

In diesem Kapitel wollen wir die in Abschnitt 3.3 angegeben Kombinationen aus Framing-Technik und Kontraktkompositionsmechanismus in Hinblick auf ihre Anwendbarkeit und ihren Nutzen für die Verifikation von Software-Produktlinien untersuchen. Dafür erläutern wir in Abschnitt 4.1 die verwendeten Fallstudien für die Evaluierung, die schon im Kontext der Kontraktkomposition ohne Framing analysiert wurden [Thüm, 2015]. In Abschnitt 4.2 untersuchen wir die Verwendung von Framing in den Fallstudien und die Anwendbarkeit der Framing-Techniken, sowie die Eigenschaften, die die Kontraktkompositionen besitzen. Anschließend vergleichen wir die Verifikation mit und ohne Framing, um festzustellen, ob der Aufwand mit Framing geringer ist und ob mehr Beweise geschlossen werden können.

4.1 Fallstudien

Um die Framing-Techniken zu untersuchen benötigen wir einige Feature-orientierte Produktlinien, um eine Aussage über die Notwendigkeit und Anwendbarkeit der einzelnen Framing-Techniken betrachten zu können. Dazu verwenden wir die gleichen Fallstudien, die bereits für die Kontraktkompositionsmechanismen ohne Framing verwendet wurden [Thüm, 2015], da wir so die Auswirkung vom Framing auf die Eigenschaften der Kontrakte und der anwendbaren Kontraktkompositionsmechanismen vergleichen können. Die Fallstudien lassen sich in drei Kategorien einteilen [Thüm, 2015]. Produktlinien, die von Grund auf entwickelt wurden (*GPL-scratch, BankAccount, IntegerList, UnionFind, StringMatcher*), existierende Programme mit Kontrakten, die in Feature-Module zerlegt wurden (*IntegerSet, Numbers, DiGraph, ExamDB, Paycard, Poker*), und Produktlinien, die im Nachhinein mit Kontrakten spezifiziert wurden (*GPL, Elevator, Email*).

Die Fallstudien wurden bereits bezüglich verschiedener Aspekte untersucht [Thüm, 2015]. Zunächst wurden die Kontrakte auf ihre Featurezugehörigkeit, also ob sie zu einem Kernfeature gehören, analysiert und wie viele Kontraktverfeinerungen und

http://dx.doi.org/10.24355/dbbs.084-201711280920

alternative Kontrakteinführungen existieren, wobei uns nur die Kontraktverfeinerungen interessieren. Die Preserving-Eigenschaften wurden für Vor- und Nachbedingungen in den Kontraktverfeinerungen untersucht, die wir überprüft haben, um sie für unsere Analyse, ob die Preserving-Eigenschaften noch gelten, wenn die Bedingungen für das Framing ergänzt werden, zu verwenden. Ebenfalls wurde ausgewertet, was sich in einer Kontraktverfeinerung ändert (Vor-, Nachbedingung, beides, nur ein Teil einer Bedingung), wobei wir hier noch zusätzlich den Frame betrachten wollen. Zuletzt wurde noch die Anwendbarkeit der Kompositionsmechanismen ohne Framing untersucht, die wir ebenfalls kontrolliert haben, um sie anschließend mit der Anwendbarkeit der Kompositionsmechanismen mit Framing vergleichen zu können.

Da die Kontrakte nicht überall den Frame spezifiziert haben, fügen wir diesen nachträglich hinzu. Dazu analysieren wir die Methode im aktuellen Feature und fügen alle Felder im `assignable` hinzu, die in der Methode benötigt werden, falls der Frame leer ist, wird er als `assignable \nothing` spezifiziert. Für die Kontraktverfeinerungen hängt es von dem Kompositionsmechanismus ab, der in einigen Methoden explizit (zum Beispiel durch `\consecutive_contract`) oder implizit (durch `original`) angegeben wird, wie der Frame aussehen muss. Für Plain Contracting und Consecutive Contract Refinement können wir einen leeren Frame als `assignable \nothing` deklarieren. Bei Contract Overriding müssen wir manuell die Frames aus den vorherigen Features ergänzen. Für Conjunctive Contract Refinement und Cumulative Contract Refinement geben wir `assignable \everything` an, damit durch den Frame Cut keine Felder verloren gehen. Und zuletzt bei Explicit Contract Refinement wird ein leerer Frame als `assignable \original` deklariert. Wenn die Methoden als `pure` deklariert worden sind, wird der Frame nicht explizit angegeben, da er dadurch ohnehin als `assignable \nothing` spezifiziert ist.

Insgesamt haben wir 14 Fallstudien, die alle in Java mit JML Kontrakten implementiert sind. Im Anhang ist der Ursprung der Produktlinien beschrieben, die auch im FeatureIDE Repository zur Verfügung stehen.[1]

4.2 Analyse von Framing in Produktlinien und Anwendbarkeit von Framing-Techniken

In diesem Kapitel befassen wir uns mit den Ergebnissen und Erkenntnissen aus der Analyse der 14 Fallstudien. Zunächst betrachten wir, wie oft der Frame mit `\nothing` spezifiziert wurde und wie oft Felder im Frame angegeben werden. Anschließend analysieren wir, wie häufig der Frame in einer Kontraktverfeinerung geändert wird, dadurch können wir eine Aussage darüber treffen, wie häufig wir für den Frame überhaupt eine Kompositionstechnik benötigen. Interessant ist außerdem noch, was für Felder in den Kontraktverfeinerungen zum Frame hinzugefügt werden, also ob das Feld in dem Feature, in dem es hinzugefügt wird, neu deklariert wurde oder ob es aus einem anderen Feature stammt. Anschließend diskutieren wir, wie

[1] https://github.com/FeatureIDE/FeatureIDE/tree/develop/plugins/de.ovgu.featureide.examples/featureide_examples

http://dx.doi.org/10.24355/dbbs.084-201711280920

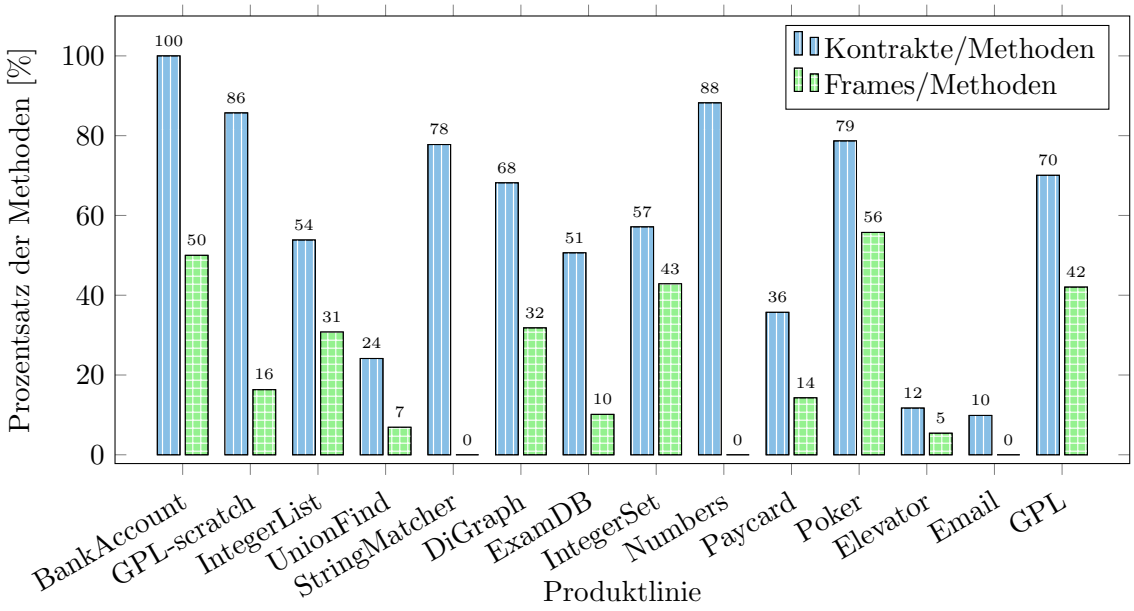

Abbildung 4.1: Prozentsatz der Kontrakte und nichtleeren Frames

sich die Eigenschaften unter Berücksichtigung des Frames ändern, da sich durch die Ergänzungen des Frames bei der Caller- und Callee-Kompatibilität einige Änderungen ergeben, und welche Kontraktkompositionsmechanismen noch anwendbar sind.

Die angegebenen Werte weichen gegebenenfalls von den ursprünglichen Werten von Thüm [Thüm, 2015] ab, da sich durch die Überarbeitung der Daten noch Änderungen ergeben haben. Außerdem enthalten die Diagramme in verschiedenen Abschnitten über 100%, da hier mehrere Eigenschaften gelten oder mehrere Techniken auf einzelne Kontraktverfeinerungen angewendet werden können.

4.2.1 Häufigkeit von Framing

Zunächst einmal interessiert es uns, wie häufig Framing in den Methoden der Fallstudien spezifiziert werden muss, das heißt, wie oft der Frame nicht leer ist. Nichtleer bedeutet in diesem Kontext, dass der Frame nicht als `assignable \nothing` spezifiziert wurde. Eigentlich ist der Defaultwert für JML `assignable \everything`, aber damit die Verifikation funktioniert haben wir bereits in Kapitel 3 festgestellt, dass wir angeben müssen, was nicht geändert wird. Die Häufigkeit ist insbesondere deshalb interessant, um festzustellen wie Häufig der Frame in Kontraktverfeinerungen geändert wird. Außerdem ist für leere Frames in Originalkontrakt und Kontraktverfeinerung die Framing-Techniken irrelevant, da der komponierte Kontrakt ebenfalls leer ist. Das kann zum einen mit allen Framing-Techniken erreicht werden (wodurch sich die verwendeten Kompositionsmechanismen nicht ändern) und zum anderen ändern sich dadurch auch die Preserving-Eigenschaften nicht für die Kontrakte, da alle Bedingungen für den Frame bei gleichen Frames (in diesem Fall `\nothing`) erfüllt sind.

http://dx.doi.org/10.24355/dbbs.084-201711280920

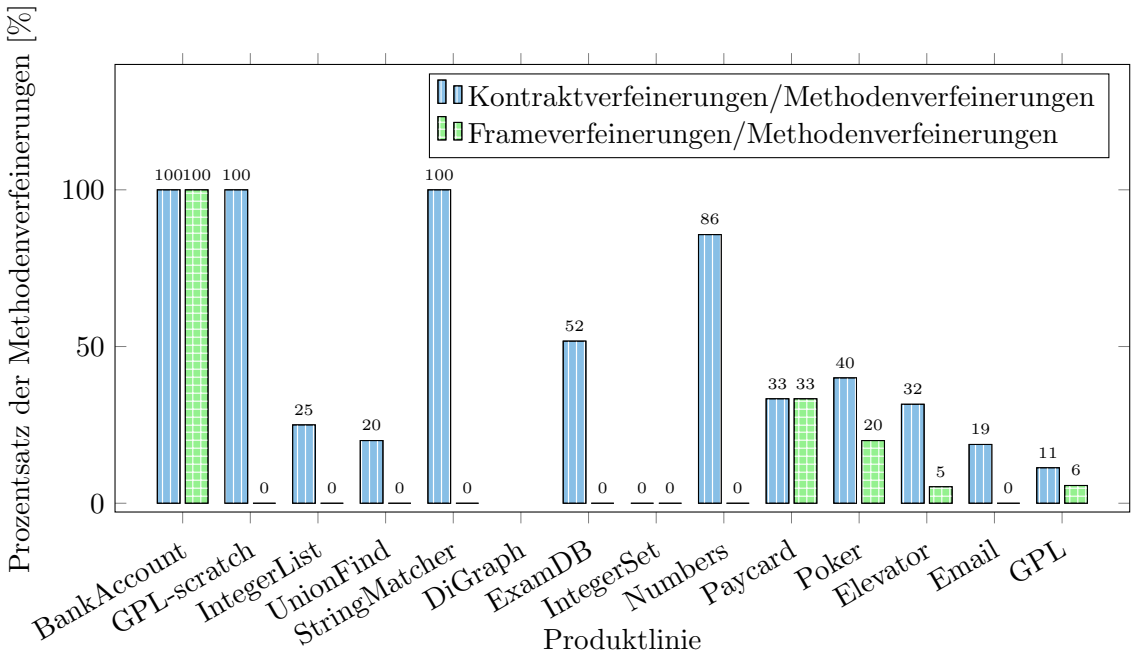

Abbildung 4.2: Prozentsatz der Kontrakt- und Frameverfeinerungen im Vergleich zur Anzahl der Methodenverfeinerungen

In Abbildung 4.1 ist dargestellt, wie viel Prozent aller Methoden einen Kontrakt haben und wie viele einen nichtleeren Frame besitzen. Der Anteil der mit Kontrakten spezifizierten Methoden liegt bei den Produktlinien zwischen 10% (*Elevator*) und 100% (*BankAccount*). Wir haben in den 14 Fallstudien insgesamt 381 Kontrakte in 559 Methoden, also im Durchschnitt haben 68% aller Methoden Kontrakte. Wir sehen, dass die Fallstudien *StringMatcher*, *Numbers* und *Email* nur leere Frames haben, während bei den anderen Fallstudien zwischen 19% (*GPL-scratch*) und 75% (*IntegerSet*) der Kontrakte einen nichtleeren Frame besitzen. Durchschnittlich spezifizieren 45% der Kontrakte in allen Fallstudien einen nichtleeren Frame, also haben wir in 171 Kontrakten einen nichtleeren Frame.

Dadurch ergibt sich, dass die Fallstudien *StringMatcher*, *Numbers* und *Email* für die weiteren Untersuchungen uninteressant sind, weshalb 11 Fallstudien verbleiben. Wie oben bereits erwähnt, ändern sich die Preserving-Eigenschaften bei leeren Frames nicht. Auch die Kompositionsmechanismen können, wie in den Untersuchungen von Thüm [Thüm, 2015] angegeben, verwendet werden, da der Frame bei `\nothing` bleibt und dafür die verwendete Framing-Technik irrelevant ist.

4.2.2 Häufigkeit von Frameverfeinerungen

Für die Kompositionsmechanismen sind vor allem die Methoden mit Kontraktverfeinerung interessant, da wir hier die Mechanismen benötigen, und für die Framing-Techniken sind die Frames der Methoden mit Kontraktverfeinerung wichtig. Mit beidem können wir den Anteil an Kontraktverfeinerungen bestimmen, für die wir

http://dx.doi.org/10.24355/dbbs.084-201711280920

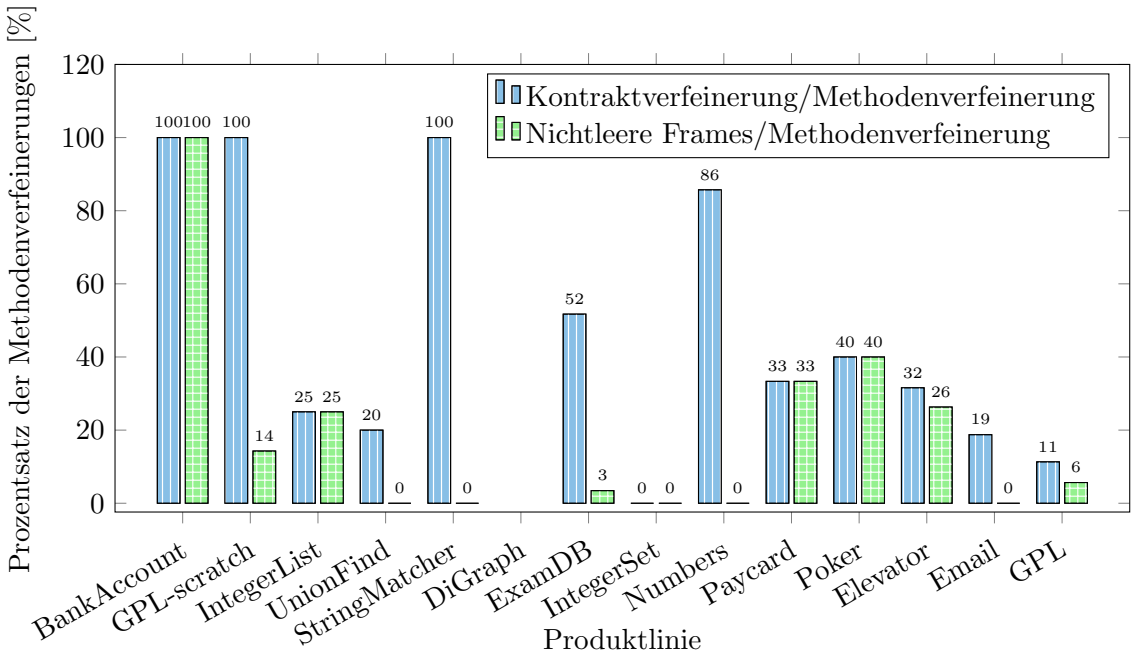

Abbildung 4.3: Prozentsatz der Kontraktverfeinerungen und nichtleeren Frames im Vergleich zur Anzahl der Methodenverfeinerungen

überhaupt eine Framing-Technik benötigen. Dafür sind die Kontraktverfeinerungen interessant, die einen nichtleeren Frame haben, da wir speziell dafür die Framing-Techniken einführen wollen. Aber auch die Methoden, deren Originalkontrakt einen nichtleeren Frame definiert, sind relevant, da auch hier die Komposition des Frames vom Kompositionsmechanismus abhängt und nicht einfach als `\nothing` angegeben werden kann. Methoden, die zwar eine Methodenverfeinerung haben, aber keine Kontraktverfeinerung benutzen, verwenden einfach Plain Contracting und Plain Framing, da es hier nicht notwendig ist, Kontraktverfeinerungen zu erlauben.

In Abbildung 4.2 sehen wir, wie viele der Methodenverfeinerungen eine Kontraktverfeinerung und eine Frameverfeinerung besitzen. *DiGraph* hat keine Methodenverfeinerungen und deswegen auch keine Kontraktverfeinerungen, weshalb in dem Diagramm kein Balken zu sehen ist. Die verbliebenen Produktlinien besitzen 3 Methodenverfeinerungen (*Paycard*) bis 53 Methodenverfeinerungen (*GPL*), insgesamt haben wir 170 Methodenverfeinerungen. *IntegerSet* besitzt zwar Methodenverfeinerungen, aber keine Kontraktverfeinerungen. Die restlichen 12 Fallstudien geben bei 11% (*GPL*) bis 100% (*BankAccount, GPL-scratch, StringMatcher*) eine Kontraktverfeinerung für die Methodenverfeinerungen an. Von allen Methodenverfeinerungen werden insgesamt nur 35% mit Kontrakten spezifiziert. Wir sehen auch, dass gerade mal fünf Fallstudien (*BankAccount, Paycard, Poker, Elevator, GPL*) den Frame in der Kontraktverfeinerung ändern und dort der Anteil an Frameverfeinerungen pro Kontraktverfeinerung zwischen 17% (*Elevator*) und 100% (*BankAccount, Paycard*) liegt. Insgesamt finden wir hier in allen Fallstudien bei 60 Kontraktverfeinerungen in 170 Methodenverfeinerungen gerade mal 11 Frameverfeinerungen.

http://dx.doi.org/10.24355/dbbs.084-201711280920

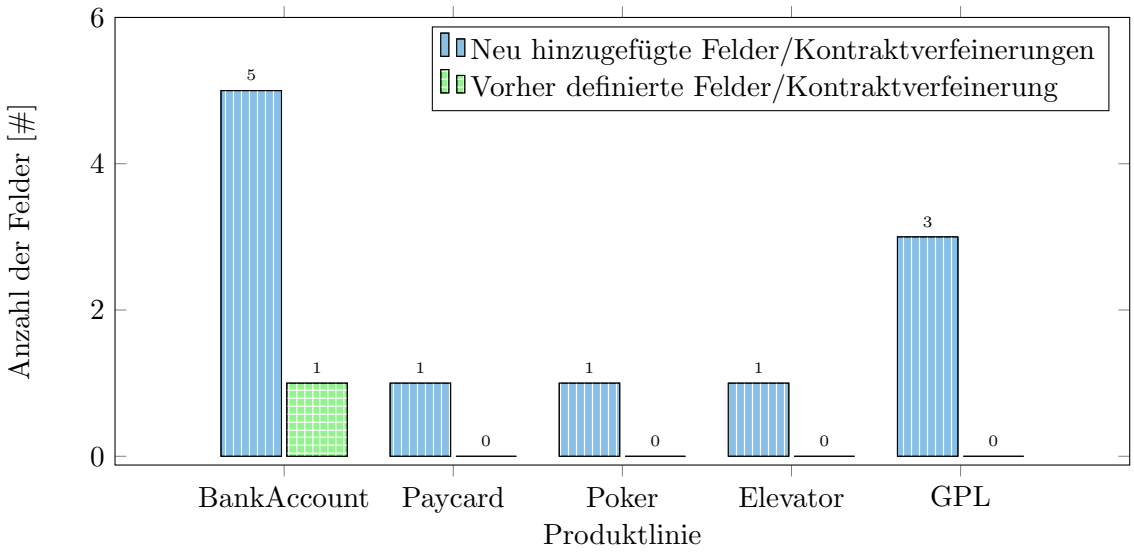

Abbildung 4.4: Eigenschaften der hinzugefügten Felder

In Abbildung 4.3 sehen wir die nichtleeren Frames pro Methodenverfeinerung, das heißt, dass entweder der Originalkontrakt oder die Kontraktverfeinerung einen nichtleeren Frame hat. Dadurch sind alle Methodenverfeinerungen aus Abbildung 4.2 enthalten und zusätzlich die Methoden, deren Originalkontrakt keinen leeren Frame hat. Wir sehen, dass sich dadurch die Werte bei den Produktlinien *GPL-scratch*, *IntegerList*, *ExamDB*, *Poker* und *Elevator* ändern. Bei fünf der Fallstudien haben wir allerdings nur leere Frames in den Kontraktverfeinerungen und bei *DiGraph* haben wir keine Methoden- und Kontraktverfeinerungen. Wir haben für die Kontraktverfeinerungen zwischen 7% (*ExamDB*) und 100% (*BankAccount, IntegerList, Paycard, Poker*) nichtleere Frames. Insgesamt sind 19 nichtleere Frames auf acht Produktlinien verteilt.

Aus Abbildung 4.3 ergibt sich, dass nur acht der Fallstudien für die weiteren Untersuchungen von Relevanz sind, da wir im Vorfeld schon festgestellt haben, dass leere Frames in Originalkontrakt und Kontraktverfeinerung weder Eigenschaften ändern noch Einfluss auf die Anwendbarkeit der Mechanismen haben. Die verbleibenden Fallstudien sind *BankAccount* (5 nichtleere Frames), *Elevator* (5 nichtleere Frames), *GPL* (3 nichtleere Frames), *Poker* (2 nichtleere Frames), *IntegerList* (1 nichtleerer Frame), *GPL-scratch* (1 nichtleerer Frame), *ExamDB*(1 nichtleerer Frame) und *Paycard*(1 nichtleerer Frame). Wegen der gerade mal 19 nichtleeren Frames (von 60 Kontraktverfeinerungen) können wir feststellen, dass nur 32% aller Kontraktverfeinerungen eine Framing-Technik benötigen. Da nur 60 von 170 Methodenverfeinerungen einen Kontrakt besitzen, können die verbliebenen 110 Methoden, wie oben bereits erläutert, einfach Plain Contracting mit Plain Framing verwenden, auf die restlichen 60 Methodenverfeinerungen können wir Plain Contracting per Definition nicht anwenden.

http://dx.doi.org/10.24355/dbbs.084-201711280920

4.2.3 Aufbau der Frameverfeinerungen

Eine weitere interessante Eigenschaft ist, was für Felder dem Frame in der Kontraktverfeinerung hinzugefügt werden. Dafür benötigen wir nur die fünf Fallstudien, die den Frame in der Kontraktverfeinerung ändern (*BankAccount, Paycard, Poker, Elevator, GPL*). Die hinzugefügten Felder teilen wir ein in Felder, die in dem Feature, in dem sie dem Frame hinzugefügt werden, neu deklariert worden sind, und Felder, die in einem anderen Feature deklariert wurden. Das ist speziell für die Datengruppen und Dynamic Frames wichtig, da hier nur neu deklarierte Felder hinzugefügt werden dürfen. Für die anderen Framing-Techniken ist es nicht weiter relevant. Die Datengruppen und Dynamic Frames können wir im Prinzip immer anwenden, wenn keine Felder aus anderen Features zum Frame hinzugefügt werden (unabhängig davon, ob die Datengruppen mit Plain Framing oder Frame Cut komponiert werden).

Wie wir in Abbildung 4.4 sehen, werden in den 11 Frameverfeinerungen 12 Felder hinzugefügt, aber es wird kein Feld aus dem Frame entfernt. In 10 Frameverfeinerungen wird nur ein Feld dem Frame hinzugefügt. Nur in einem einzigen Fall werden zwei Felder dem Frame hinzugefügt in der Produktlinie *BankAccount* in der Methode `Application.nextYear()`. In den Frameverfeinerungen wird ebenfalls nur einmal ein Feld hinzugefügt, das nicht in dem Feature der Kontraktverfeinerung definiert wurde, und zwar ebenfalls in der Methode `Application.nextYear()` aus der Fallstudie *BankAccount*. Die Methode zeigen wir in Quelltext 4.1 mit dem Originalkontrakt im Feature *BankAccount* und in Quelltext 4.2 mit der Kontraktverfeinerung im Feature *Interest*. Beide Felder beziehen sich hier auf Felder von `account`, wobei das Feld `balance` im Feature *BankAccount* definiert wurde und das Feld `interest` im Feature *Interest*.

feature module *BankAccount*

```
1  public class Application {
2      Account account = new Account();
3      /*@
4        @ requires true;
5        @ ensures true;
6        @ assignable \nothing;
7        @*/
8      void nextYear() {
9      }
10     ...
11 }
```

Quelltext 4.1: Methode `nextYear` im Feature *BankAccount*

Letztlich heißt das für uns, dass wir in 92% der Kontraktverfeinerungen Datengruppen und Dynamic Frames verwenden können. Nur die *BankAccount* Produktlinie können wir nicht vollständig mit Datengruppen oder Dynamic Frames ausdrücken.

4.2.4 Granularität der Kontraktverfeinerungen

Die Granularität der Kontraktverfeinerungen ist ebenfalls ein interessanter Punkt bei der Analyse der Fallstudien. Dazu betrachten wir, was sich in den Kontraktver-

http://dx.doi.org/10.24355/dbbs.084-201711280920

```
1  class Application {                                    feature module Interest
2      ...
3      /*@ \consecutive_contract
4        @
5        @ ensures account.balance == \old(account.balance)
6        @ + \old(account.interest) && account.interest == 0;
7        @ assignable account.interest, account.balance;
8        @*/
9      void nextYear() {
10         original();
11         account.balance += account.interest;
12         account.interest = 0;
13     }
14
15 }
```

Quelltext 4.2: Methode `interest` im Feature *Interest*

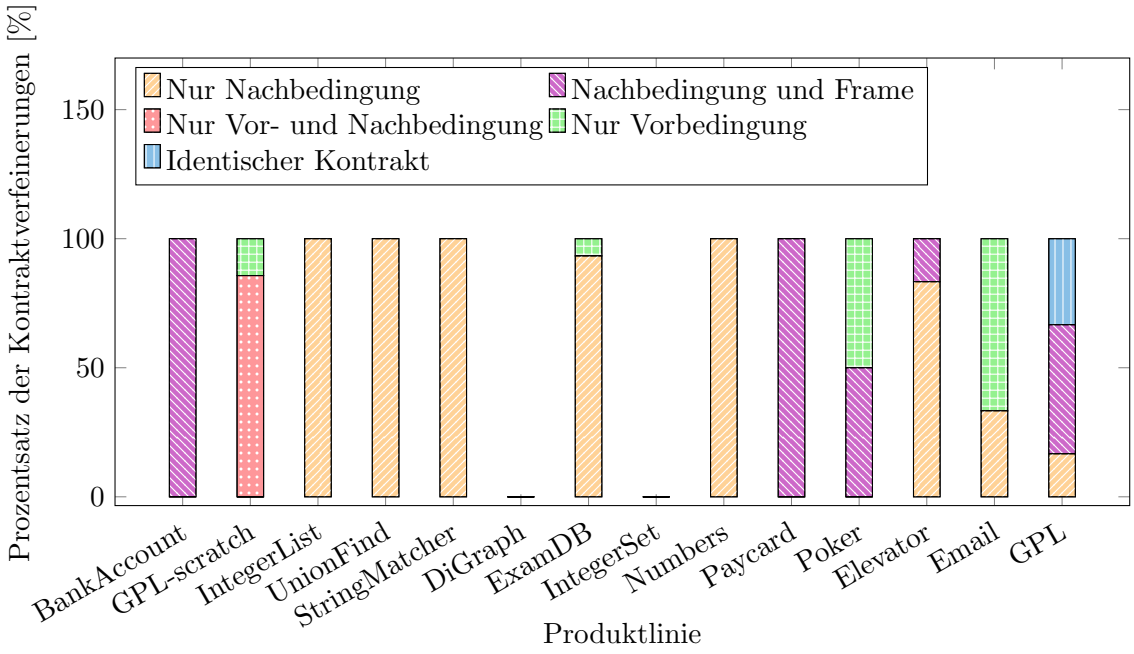

Abbildung 4.5: Granularität der Änderungen in Kontraktverfeinerungen (andere Kombinationen als die angegebenen sind nicht vorhanden)

feinerungen verändert, also Vor-, Nachbedingung und Frame, und in welcher Konstellation die Änderungen auftauchen (zum Beispiel Vor- und Nachbedingung, nur Frame).

In Abbildung 4.5 stellen wir dar, wie die Kontraktverfeinerungen konkret aussehen. *DiGraph* und *IntegerSet* haben keine Kontraktverfeinerungen und deswegen keine Balken. Wir sehen, dass am häufigsten die Nachbedingung geändert wird (88%), der Frame (18%) und die Vorbedingung (18%) werden dagegen nur selten geändert. Die Änderungen der Nachbedingungen teilen sich nach Vor- und Nachbedingung (11%), Frame und Nachbedingung (21%) und nur die Nachbedingung (68%) auf. Die Vor-

http://dx.doi.org/10.24355/dbbs.084-201711280920

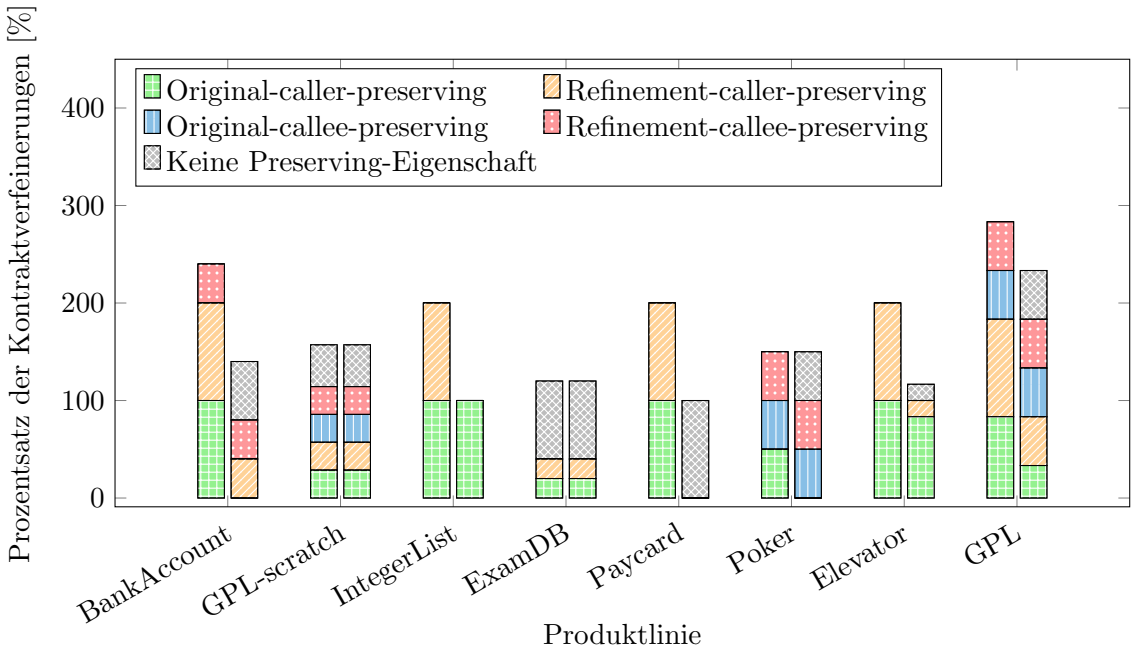

Abbildung 4.6: Preserving-Eigenschaften der Kontraktverfeinerungen ohne (links) und mit (rechts) Berücksichtigung des Frames

bedingung wird alleine (45%) oder mit der Nachbedingung (55%) verfeinert. Der Frame wird immer nur mit der Nachbedingung zusammen geändert. Nur in *GPL* haben wir Kontrakte, die in der Kontraktverfeinerung identisch sind.

Wir sehen also, dass wenn der Frame geändert wurde, auch die Nachbedingung geändert wurde. Das können wir recht einfach begründen, denn wenn wir ein neues Feld ändern, wird meistens auch angegeben, was bei der Ausführung der Methode mit dem Feld gemacht wurde. Andersrum betrachtet, wenn in der Nachbedingung ein neues Feld geändert wird, muss es dem Frame hinzugefügt werden, da ansonsten der Kontrakt nicht valide ist.

Ohne den Frame konnte, wenn nur die Nachbedingung geändert wurde, Cumulative Contract Refinement, Conjunctive Contract Refinement und Consecutive Contract Refinement analog verwendet werden, da bei allen nur Änderungen durch die Vorbedingung entstehen [Thüm, 2015]. Wenn wir den Frame zusätzlich ändern, können wir nur noch Cumulative Contract Refinement und Conjunctive Contract Refinement analog verwenden, da beide Frame Cut verwenden. Consecutive Contract Refinement haben wir mit Spezifikationsfällen umgesetzt, die nicht kompatibel zum Frame Cut sind, der Extended Frame Cut wäre es gewesen, aber der lässt sich mit JML nicht umsetzten, da dort nur Felder und keine Mengen im Frame angegeben werden können.

4.2.5 Kontraktkompatibilität der Produktlinien

In Abschnitt 3.1.1 haben wir besprochen, wie sich die Definitionen von den Preserving-Eigenschaften der Kontraktkompositionen ändert, wenn wir Framing zu den Vor-

http://dx.doi.org/10.24355/dbbs.084-201711280920

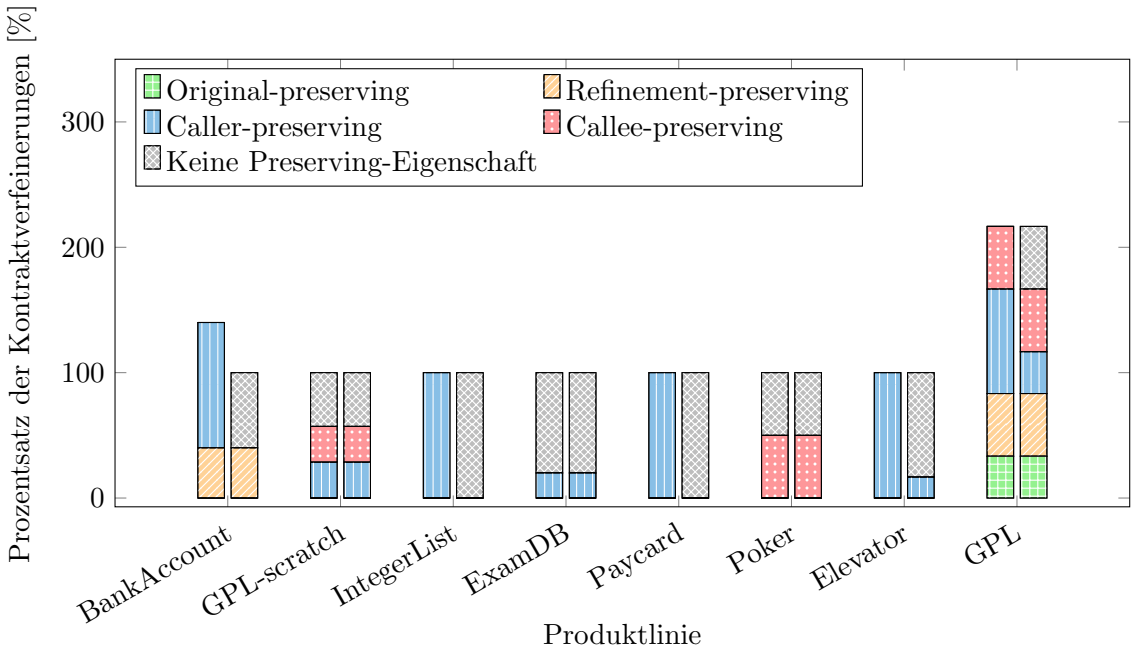

Abbildung 4.7: Kombinierte Preserving-Eigenschaften der Kontraktverfeinerungen ohne (links) und mit (rechts) Berücksichtigung des Frames

und Nachbedingungen hinzufügen. Interessant ist hierbei, wie sich das Framing auf die Preserving-Eigenschaften der Fallstudien auswirkt. Da wir festgestellt haben, dass nur in acht Fallstudien Framing überhaupt eine Rolle in Kontraktverfeinerungen spielt, betrachten wir die anderen sechs Fallstudien hier nicht (*UnionFind, StringMatcher, DiGraph, IntegerSet, Numbers, Email*). Bei den verbliebenen acht Fallstudien werden für die Eigenschaften aber alle Kontraktverfeinerungen berücksichtigt, auch die mit leeren Frames. Die Analyse basiert auf der Komposition von zwei Kontrakten und erhält eine Eigenschaft, wenn sie für alle Produkte gilt, da sich die Eigenschaften in den Produkten teilweise unterscheiden. Außerdem hängen die Eigenschaften von dem verwendeten Kompositionsmechanismus ab, der, wie in Abschnitt 4.1 bereits erwähnt, in den Kontrakten angegeben wird.

In Abbildung 4.6 sehen wir, wie sich die Preserving-Eigenschaften ändern, wenn wir Framing bei original-caller-, original-callee-, refinement-caller- und refinement-callee-preserving berücksichtigen. Der linke Balken stellt die Preserving-Eigenschaften ohne Framing dar und der rechte Balken die Preserving-Eigenschaften mit Framing. Wir sehen, dass sich original-callee-preserving und refinement-callee-preserving nicht verändert haben. Original-callee-preserving verbleibt bei 14% und refinement-callee-preserving bei 19% in allen Kontraktverfeinerungen der acht Fallstudien. Die Caller-Kompatibilität dagegen weist Änderungen auf. Drei Fallstudien sind im Vergleich zu vorher in keinem Fall mehr original-caller-preserving, bei drei Fallstudien bleibt original-caller-preserving gleich. Bei *Elevator* sinkt original-caller preserving von 100% auf 83% und bei *GPL* von 83% auf 33%. Refinement-caller-preserving ändert sich nur in *GPL-scratch* und *ExamDB* nicht und in *Poker* war von vornherein kein refinement-caller-preserving vorhanden. In *IntegerList* und *Paycard* ver-

http://dx.doi.org/10.24355/dbbs.084-201711280920

schwindet refinement-caller-preserving ganz. Bei *BankAccount* sind nur noch 40% statt 100% refinement-caller-preserving, bei *Elevator* und *GPL* sind es 17% statt 100%. Durchschnittlich sind ohne Framing 56% der Kontraktverfeinerungen aus den acht Produktlinien original-caller-preserving und refinement-caller-preserving. Mit Framing haben sind nur noch 30% original-caller-preserving und 26% refinement-caller-preserving. Nur in *IntegerList* gibt es keine Kontraktverfeinerung, die keine Preserving-Eigenschaften bieten egal ob mit oder ohne Framing. Bei *GPL-scratch* und *ExamDB* bleibt der Anteil an Kontraktverfeinerungen mit Framing gleich (43% und 80%). In den anderen fünf Produktlinien haben ohne Framing alle Kontraktverfeinerungen Preserving-Eigenschaften und mit Framing haben 17% (*Elevator*) bis 100% (*Paycard*) keine Preserving-Eigenschaften mehr.

In Abbildung 4.7 sind die Änderungen für die kombinierten Preserving-Eigenschaften dargestellt, wenn Framing hinzugefügt wird. Original-preserving ist nur in der Produktlinie *GPL* vorhanden und bleibt bei 33% in der Produktlinie und bei 5% in allen Kontraktverfeinerungen der acht Fallstudien. Refinement-preserving war nur in *GPL* und *BankAccount* gegeben. In der *GPL* Produktlinie bleibt refinement-preserving bei 50% und in *BankAccount* bei 40%. Insgesamt bleibt das refinement-preserving in den acht Produktlinien bei 12%. Caller-preserving bleibt nur in *ExamDB* mit 20% gleich und *Poker* hatte keine Kontraktverfeinerungen, die caller-preserving sind. Vier Produktlinien (*BankAccount, GPL-scratch, IntegerList, Paycard*) haben mit Framing gar keine Kontraktverfeinerungen mehr, die caller-preserving sind, und in *Elevator* bleiben 17% statt 100% und in *GPL* bleiben 33% statt 83% caller-preserving. Insgesamt haben wir in allen Kontraktverfeinerungen der acht Fallstudien ohne Framing 53% und mit Framing 19%, die caller-preserving sind. Callee-preserving ändert sich in gar keiner Produktlinie. In *GPL-scratch*, *ExamDB* und *Poker* bleibt der Anteil an Kontraktverfeinerungen, die keine kombinierten Preserving-Eigenschaften haben, gleich. In den anderen Fallstudien erhöht sich der Anteil zwischen 50% (*GPL*) und 100% (*IntegerList, Paycard*).

Die Eigenschaften original-callee-preserving und refinement-callee-preserving ändern sich nicht. Das lässt sich mit den Bedingungen für den Frame begründen, der Originalframe bei original-callee-preserving beziehungsweise die Frameverfeinerung bei refinement-callee-preserving muss eine Teilmenge des komponierten Frames sein. Das heißt wir können den komponierten Frame gegenüber dem Originalframe und der Frameverfeinerung beliebig erweitern, aber nicht verringern. Da der Frame in den Produktlinien nicht einmal verringert wird, sondern nur erweitert, kann sich die Callee-Kompatibilität nicht ändern.

Die Häufigkeit von original-caller-preserving und refinement-caller-preserving reduziert sich durch das Framing. Das können wir ebenfalls mit den Bedingungen für den Frame begründen. Für original-caller-preserving muss der komponierte Frame eine Teilmenge des Originalframes sein. Also dürfen in dem verfeinerten Frame keine Felder hinzugefügt werden. Analog gilt für refinement-caller-preserving, dass der komponierte Frame Teilmenge der Frameverfeinerung ist. Da 11 mal der Frame verfeinert wird, ist in diesen 11 Fällen die Kontraktverfeinerung nicht original-caller-

http://dx.doi.org/10.24355/dbbs.084-201711280920

preserving, die ohne Berücksichtigung des Frames alle original-caller-preserving waren. In 13 Fällen werden im Originalframe ein oder mehr Felder angegeben, die in der Frameverfeinerung nicht auftauchen, aber im komponierten Frame und deshalb sind diese Fälle nicht mehr refinement-caller-preserving.

Aus den Erkenntnissen für die grundlegenden vier Eigenschaften, lassen sich die Änderungen durch den Frame in den kombinierten Eigenschaften erklären. Callee-preserving bleibt erhalten, da sich weder original-callee-preserving noch refinement-callee-preserving geändert hat. Caller-preserving wird von original-caller-preserving und refinement-caller-preserving beeinflusst, weshalb hier bis zu 24 Fälle nicht mehr caller-preserving sind, da sich die Fälle, die nicht mehr original-caller-preserving oder refinement-caller-preserving sind, überschneiden können. Insgesamt sind davon 15 Fälle betroffen. Die zwei Fälle, die original-preserving sind, haben im Originalkontrakt und in der Kontraktverfeinerung einen leeren Frame, weshalb dort keine Änderungen auftreten. Refinement-preserving bleibt in *BankAccount* erhalten, da bei beiden Kontraktverfeinerungen, die refinement-preserving sind, der Originalkontrakt einen leeren Frame hat und der komponierte Frame mit dem verfeinertem Frame identisch ist, und bei *GPL* ebenfalls, da die betroffenen Kontraktverfeinerungen leere Frames haben.

Insgesamt erhalten wir ein anderes Bild, was die Häufigkeit der Preserving-Eigenschaften über alle 14 Fallstudien betrifft. Die meisten Kontraktverfeinerungen (62%) haben keine der kombinierten Eigenschaften. Nur noch 27% sind caller-preserving. Callee-preserving (12%), original-preserving (3%) und refinement-preserving (8%) sind kaum vorhanden.

4.2.6 Anwendbarkeit

In diesem Abschnitt wollen wir untersuchen, in wie weit die Framing-Techniken auf die Fälle der Fallstudien anwendbar sind in den acht Fallstudien (unabhängig vom verwendeten Kompositionsmechanismus), die keinen leeren Frame haben, und wie sich das auf die Anwendbarkeit der Kompositionsmechanismen in allen Kontraktverfeinerungen in den 14 Fallstudien auswirkt. Denn bei den Kontraktverfeinerungen, die sowohl in dem Originalkontrakt oder der Kontraktverfeinerung einen leeren Frame haben, ist es egal, welche der Techniken wir anwenden, da der komponierte Frame immer leer ist. Bei Datengruppen und Dynamic Frames kann hier die Datengruppe oder der Dynamic Frame einfach weggelassen werden, um Spezifikationsaufwand zu vermeiden. Bei den Methoden mit nichtleerem Frame bezeichnen wir eine Framing-Technik als anwendbar, wenn wir mit der Technik in dem komponierten Kontrakt einen gültigen Frame erzeugen können. Außerdem darf keine Kopie der Spezifikation vorliegen, es sei denn, dass der komponierte Frame ohne Kopie von keiner Technik ausgedrückt werden kann.

In Abbildung 4.8 zeigen wir, wie oft sich die Framing-Techniken auf die Kontraktverfeinerungen mit nichtleeren Frames anwenden lässt. Datengruppen und Dynamic Frames können am häufigsten angewandt werden (95%), aber auch Explicit Frame

http://dx.doi.org/10.24355/dbbs.084-201711280920

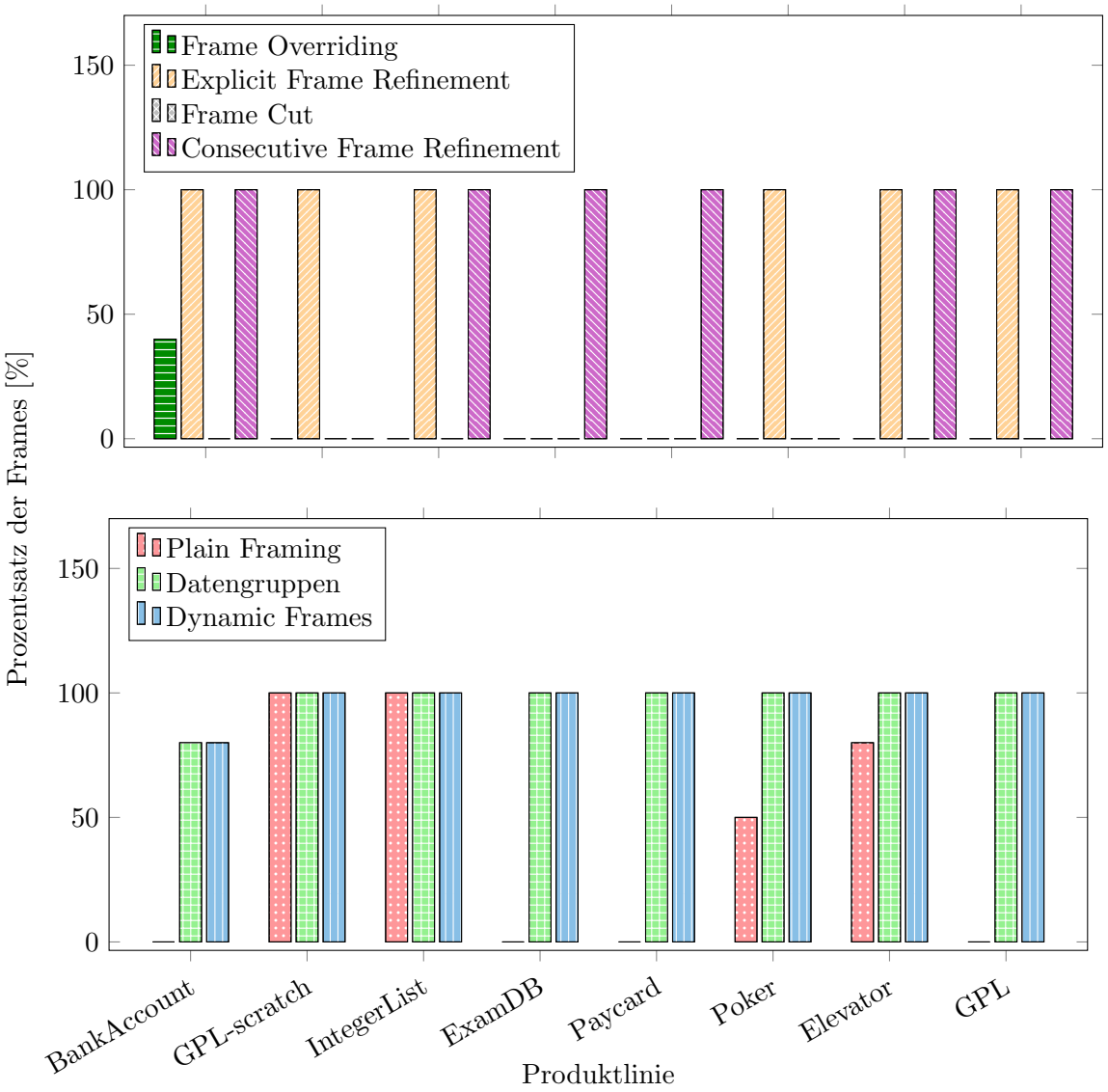

Abbildung 4.8: Anwendbarkeit der Framing-Techniken

Refinement (89%) und Consecutive Frame Refinement (84%) kann sehr häufig verwendet werden. Plain Framing (37%) und Frame Overriding (11%) kann nur selten angewendet werden und Frame Cut lässt sich gar nicht verwenden.

Die anwendbaren Techniken lassen sich leicht mit dem Aufbau der Frames erklären. Datengruppen und Dynamic Frames lassen sich nur nicht verwenden, wenn Felder hinzugefügt werden, die nicht in dem aktuellen Feature deklariert worden sind, und das kam in den Fallstudien nur einmal vor. Explicit Frame Refinement lässt sich nahezu immer anwenden, aber sobald Spezifikationsfälle in dem Originalkontrakt verwendet werden, ist es ungewiss auf welchen der Spezifikationsfälle sich das `original` bezieht, das ist nur zwei mal der Fall, einmal in der Produktlinie *ExamDB* in der Methode `addStudent()` und in der Produktlinie *Paycard* in der Methode `charge()`. Plain Framing können wir nur verwenden, wenn keine Frameverfeinerung vorliegt, das sind acht Fälle, aber einer verwendet Spezifikationsfälle, wodurch fraglich wäre, auf welchen Frame sich die neue Spezifikation bezieht. Für Frame Overriding und

http://dx.doi.org/10.24355/dbbs.084-201711280920

Frame Cut würden wir in den meisten Fällen die Spezifikation kopieren müssen, weshalb die Techniken selten anwendbar sind. Frame Overriding lässt sich nur bei den zwei Fällen anwenden, die im Originalkontrakt einen leeren Frame haben.

Wir sehen also, dass insbesondere Spezifikationsfälle Schwierigkeiten bei der Anwendung von Framing-Techniken verursachen. Hier können wir nur mit Consecutive Frame Refinement, Datengruppen und Dynamic Frames arbeiten. Wenn zusätzlich noch Felder aus anderen Features hinzugefügt werden, bleibt nur noch Consecutive Frame Refinement. Das liegt insbesondere daran, dass irgendwie bestimmt werden müsste, auf welchen der vorherigen Frames sich der neue Frame beziehen würde.

Beispiel 4.1. Dazu können wir uns die Methode `charge()` aus der Produktlinie *Paycard* ansehen. Wir haben in dem Originalkontrakt im Feature *Paycard* in Quelltext 4.3 drei Spezifikationsfälle und die Kontraktverfeinerung im Feature *LockOut* in Quelltext 4.4 bezieht sich wegen der Vorbedingung offensichtlich auf den zweiten Spezifikationsfall. Das können wir aber in Plain Framing, Frame Overriding, Explicit Frame Refinement und Frame Cut nicht ausdrücken. Für Datengruppen und Dynamic Frames müssen wir pro Spezifikationsfall einfach eine Datengruppe oder einen Dynamic Frame erstellen und in Spezifikationsfällen wird die Kontraktverfeinerung einfach als neuer Spezifikationsfall angehängt. Plain Framing würde ohnehin nur partiell für den ersten und dritten Spezifikationsfall funktionieren, da wir den Frame im zweiten Spezifikationsfall verfeinern. Ebenso würde Frame Overriding nicht funktionieren, da wir zum einen die Frames aus Spezifikationsfall eins und drei beibehalten wollen und wir im Zweiten den Frame aus dem Originalkontrakt kopieren müssten. Wegen der Kopien wäre auch Frame Cut nicht anwendbar. Explicit Frame Refinement könnten wir anwenden, wenn man einen optionalen Parameter für das `original` einführen würden, der angibt, welcher Spezifikationsfall verfeinert werden soll (was auch eine Möglichkeit für Vor- und Nachbedingung wäre), was bereits als Lösung für Spezifikationsfälle im Explicit Contract Refinement für Vor- und Nachbedingung vorgestellt wurde [Benduhn, 2012].

In Abbildung 4.9 sehen wir, wie sich die Anwendbarkeit der Kontraktkompositionsmechanismen in den acht Fallstudien mit nichtleeren Frames ändert, wenn wir die in Abschnitt 3.1 vorgestellten Framing-Techniken für den jeweilige Kompositionsmechanismus verwenden, abgesehen von Plain Contracting, da wir Plain Contracting per Definition nicht bei Kontraktverfeinerungen verwenden können. Wie wir sehen, ändert sich die Anwendbarkeit von Contract Overriding (49%), Explicit Contract Refinement (91%) und Consecutive Contract Refinement (49%) nicht. Dagegen verringert sich die Anwendbarkeit von Conjunctive Contract Refinement (von 53% auf 16%) und Cumulative Contract Refinement (von 40% auf 7%).

Die Ergebnisse für Contract Overriding, Explicit Contract Refinement und Consecutive Contract Refinement sind anhand der Framing-Technik leicht nachvollziehbar. Alle Framing-Techniken funktionieren analog zu den Techniken für Vor- und Nachbedingungen, weshalb die Framing-Techniken sich immer dann anwenden lassen, wenn der Kompositionsmechanismus für Vor- und Nachbedingung anwendbar ist.

http://dx.doi.org/10.24355/dbbs.084-201711280920

feature module *Paycard*

```
1   /*@ \consecutive_contract
2     @ public normal_behavior
3     @ requires amount>0;
4     @ {|
5     @     requires amount + balance < limit && isValid() == true;
6     @     ensures \result == true;
7     @     ensures balance == amount + \old(balance);
8     @     assignable balance;
9     @     also
10    @     requires amount + balance >= limit || isValid() == false;
11    @     ensures \result == false;
12    @     assignable \nothing;
13    @ |}
14    @
15    @ also
16    @
17    @ public exceptional_behavior
18    @ requires amount <= 0;
19    @*/
20  public boolean charge(int amount) throws IllegalArgumentException {
21    if (amount <= 0) {
22      throw new IllegalArgumentException();
23    }
24    if (this.balance+amount<this.limit && this.isValid()) {
25      this.balance=this.balance+amount;
26      return true;
27    } else {
28      return false;
29  }
```

Quelltext 4.3: Methode `charge` im Feature *Paycard*

Für Frame Cut, den wir in Conjunctive und Cumulative Contract Refinement anwenden, haben wir direkt festgestellt (siehe Abschnitt 3.1.5), dass sich, wenn überhaupt, die Framing-Technik nur mit Spezifikationskopien anwenden lässt. Da wir bereits in Abbildung 4.8 gesehen haben, dass sich damit Frame Cut nicht anwenden lässt, ist klar, dass sich die Kompositionsmechanismen in Kontraktverfeinerungen mit nichtleerem Frame (in Originalkontrakt oder Kontraktverfeinerung) nicht mehr verwenden lassen.

Nun betrachten wir die Änderungen über alle Kontraktverfeinerungen in den 14 Fallstudien. Explicit Contract Refinement lässt sich am häufigsten anwenden (92%). Die anderen Kompositionsmechanismen lassen sich bei weniger als der Hälfte aller Kontraktverfeinerungen anwenden. Consecutive Contract Refinement (43%), Contract Overriding (37%), Conjunctive Contract Overriding (32%, ohne Framing 58%) und Cumulative Contract Refinement (23%, ohne Framing 47%) decken nicht annähernd so viele Fälle ab, wie Explicit Contract Refinement.

Bei den bisherigen Ergebnissen bezüglich der Anwendbarkeit der Kompositionsmechanismen mit Framing sind Datengruppen und Dynamic Frames nicht berücksichtigt. Wir haben gesehen, dass für die Anwendbarkeit von Explicit Contract Re-

http://dx.doi.org/10.24355/dbbs.084-201711280920

feature module *LockOut*

```
1   /*@ public normal_behavior
2     @ requires amount>0;
3     @ requires amount + balance >= limit || isValid() == false;
4     @ ensures unsuccessfulOperations ==
5     @             \old(unsuccessfulOperations) + 1;
6     @ assignable unsuccessfulOperations;
7     @*/
8   public boolean charge(int amount) throws IllegalArgumentException {
9     boolean success = original(amount);
10    if (!success)
11      this.unsuccessfulOperations++;
12    return success;
13  }
```

Quelltext 4.4: Methode `charge` im Feature *LockOut*

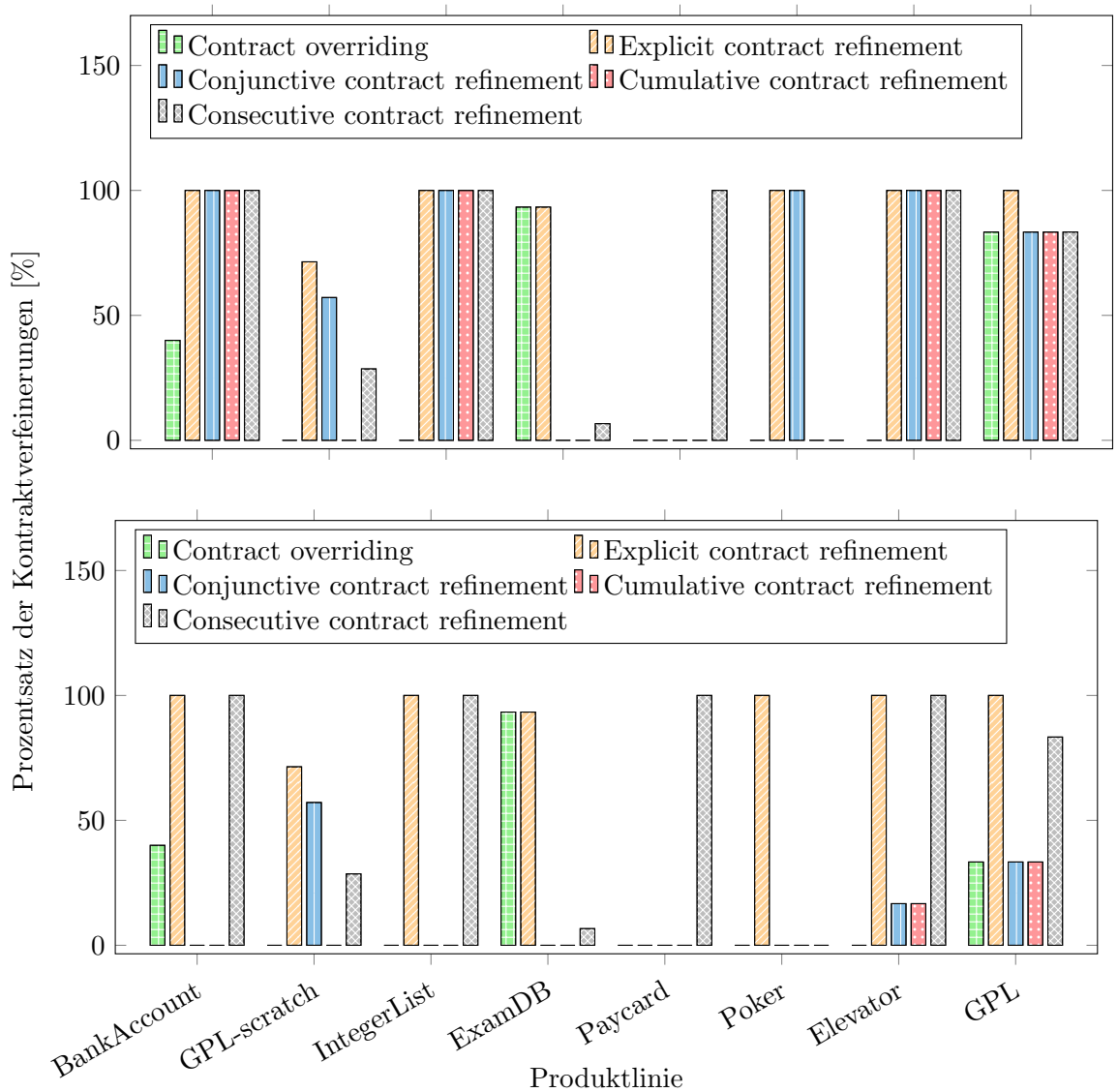

Abbildung 4.9: Anwendbarkeit der Kontraktkompositionsmechanismen mit (unten) und ohne (oben) Berücksichtigung des Frames

http://dx.doi.org/10.24355/dbbs.084-201711280920

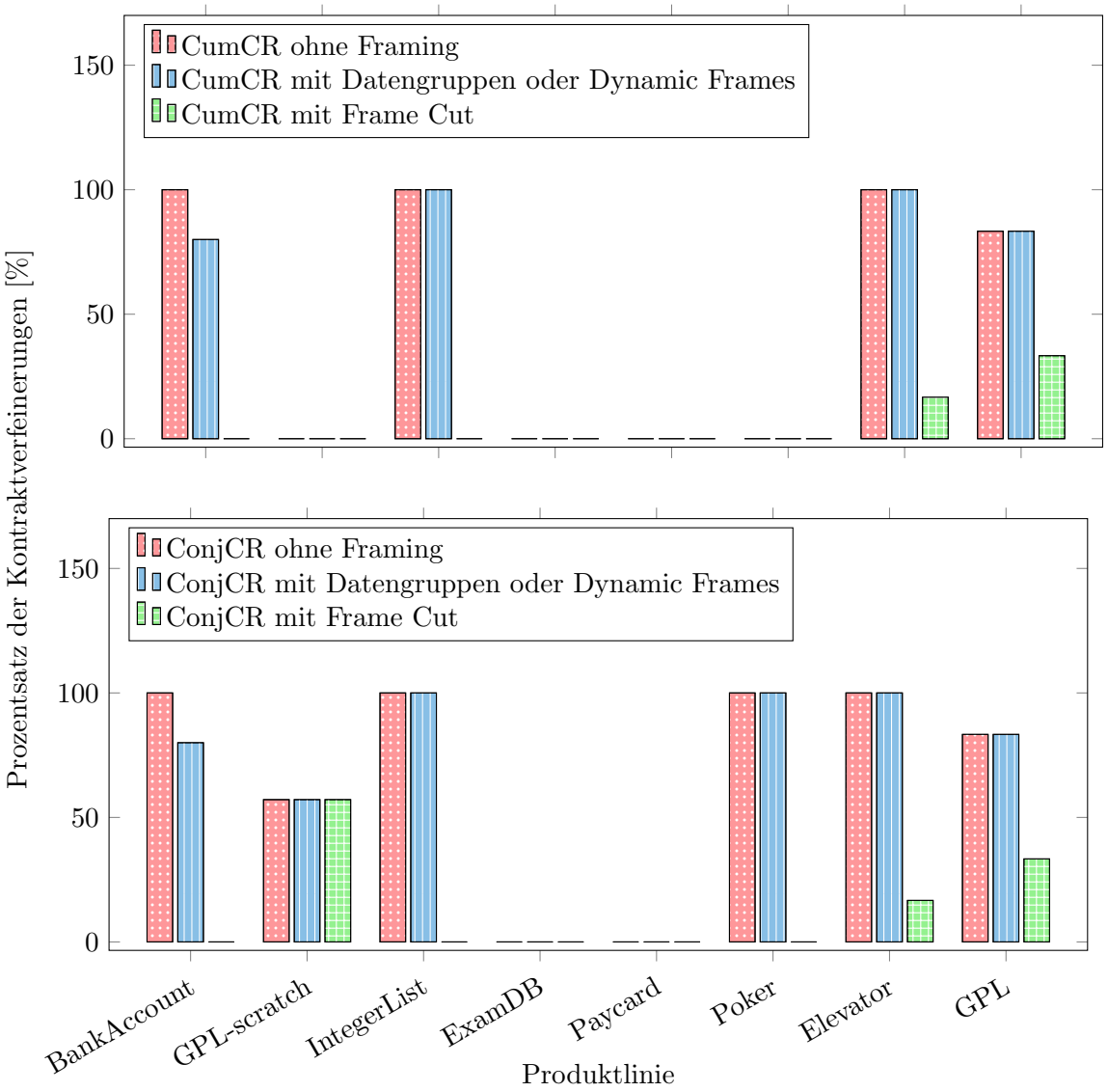

Abbildung 4.10: Cumulative Contract Refinement (oben) und Conjunctive Contract Refinement (unten) ohne Framing (links), mit Datengruppen und Dynamic Frames (mitte) und Frame Cut (rechts)

finement, Contract Overriding und Consecutive Contract Refinement die in Abschnitt 3.1 vorgestellten Framing-Techniken ausreichen und den Kompositionsmechanismus in der Anwendbarkeit nicht einschränken. Deshalb wäre in diesen Kompositionsmechanismen die Verwendung von Datengruppen oder Dynamic Frames von Nachteil, da beides einen größeren Spezifikationsaufwand bedeutet, als die in Abschnitt 3.1 verwendeten Framing-Techniken. Dagegen beeinflusst der Frame Cut im Conjunctive Contract Refinement und im Cumulative Contract Refinement die Anwendbarkeit von den Kompositionsmechanismen im negativen Sinne, deshalb betrachten wir, ob sich der Einsatz von Datengruppen oder Dynamic Frames lohnen würde.

In Abbildung 4.10 betrachten wir dementsprechend, welchen Vorteil wir mit Datengruppen und Dynamic Frames in Cumulative Contract Refinement (oberes Dia-

http://dx.doi.org/10.24355/dbbs.084-201711280920

gramm) und in Conjunctive Contract Refinement (unteres Diagramm) haben. Dafür benötigen wir wieder erst mal nur die acht Fallstudien mit nichtleeren Frames, da nur hier Änderungen auftreten. Wie wir sehen, können wir mit Datengruppen und Dynamic Frames nur in *BankAccount* nicht alle Konktraktverfeinerungen ausdrücken, die ohne Framing möglich waren. Also können wir in den acht Produktlinien mit Cumulative Contract Refinement mit Datengruppen und Dynamic Frames 37% und mit Conjunctive Contract Refinement mit Datengruppen und Dynamic Frames 51% ausdrücken. Insgesamt erhalten wir über die 14 Fallstudien so für Cumulative Contract Refinement mit Datengruppen und Dynamic Frames 45% und für Conjunctive Contract Refinement mit Datengruppen und Dynamic Frames 57%. Deshalb würde sich hier gegebenenfalls der höhere Spezifikationsaufwand lohnen.

4.3 Nutzen für die Verifikation

Im folgenden Abschnitt untersuchen wir den Einfluss von Framing auf die Verifikation. Damit wollen wir zeigen, ob Framing notwendig ist, um Beweise mit Kontrakten durchzuführen, und ob Framing die Effizienz der Verifikation erhöht. Dafür betrachten wir verschiedene Parameter, zum Beispiel die Zeit, die für die Beweise gemessen werden können. Wir verwenden für die Verifikation die Produktlinie *BankAccount*, da sie bereits verifiziert wurde und es ein zu hoher Aufwand ist, eine Produktlinie neu zu beweisen, und die einzige andere verifizierte Produktlinie ist *StringMatcher* [Thüm, 2015], die kein Framing verwendet (`assignable \nothing`). Die Produktlinie *BankAccount* verifizieren wir mit Framing und ohne Framing, damit wir anschließend feststellen können, ob mit Framing mehr Beweise geschlossen werden können und ob der Verifikationsaufwand niedriger ist.

Wir verwenden Variablitiy Encoding, um die Redundanz in der Verifikation möglichst gering zu halten [Thüm et al., 2012, Thüm et al., 2014], welches wir in Abschnitt 4.3.1 näher erläutern. Anschließend erläutern wir in Abschnitt 4.3.2, wie die Kontraktkompositionsmechanismen bisher implementiert wurden und welchen Teil der Implementierung wir verwenden. Danach beschreiben wir in Abschnitt 4.3.3 die Vorgehensweise bei der Verifikation mit und ohne Framing, die mit dem Tool KeY, das für die deduktive Verifikation entwickelt wurde, durchgeführt wird. Zuletzt beschreiben und diskutieren wir die Ergebnisse der Verifikation in Abschnitt 4.3.4.

4.3.1 Variability Encoding

Für die Verifikation wollen wir Variability Encoding und deduktive Verifikation verwenden, da wir so zum einen Kontrakte für die Verifikation benutzen können und zum anderen durch das Variability Encoding weniger Redundanz bei der Verifikation von Produktlinien haben [Thüm et al., 2012, Thüm et al., 2014].

Deduktive Verifikation verwendet Logik (oftmals First-order Logic), um festzulegen, dass ein System eine bestimmte Eigenschaft erfüllt und beweist diese mit logischen Folgerungen [Beckert and Hähnle, 2014]. Eine Möglichkeit ist es eine, in das Programm eingebettete, Spezifikation zu verwenden. In unserem Fall können wir für die

http://dx.doi.org/10.24355/dbbs.084-201711280920

deduktive Verifikation Kontrakte verwenden.

Variability Encoding beschreibt einen Transformationsprozess, in dem die Variabilität der Produktlinie zum Generierungszeitpunkt in Variabilität zur Laufzeit umgewandelt wird [Apel et al., 2013c]. Ein Produkt, das die Laufzeitvariabilität beinhaltet und alle Produkte simulieren kann, wird als *Produktsimulator* oder *Metaprodukt* bezeichnet [Apel et al., 2013c, Thüm et al., 2012]. Dafür werden alle Features komponiert und für jedes Feature wird eine boolesche Variable angelegt, die die Auswahl des Features repräsentiert [Apel et al., 2013c]. Für jede Methodenverfeinerung wird eine Dispatcher-Methode generiert, die die Methodenverfeinerung ausführt, falls das Feature, zu dem die Verfeinerung gehört, ausgewählt ist (also die boolesche Variable mit `true` belegt ist). Die Abhängigkeiten zwischen den Features wird mithilfe einer logischen Formel über die booleschen Variablen der Features dargestellt, die die Ausführung invalider Produktkonfigurationen verhindert.

Für die Spezifikation des Metaproduktes im Variability Encoding wird eine *Metaspezifikation* generiert, indem die Spezifikation der Feature Module abhängig von den booleschen Variablen der Features im Metaprodukt ergänzt werden [Thüm et al., 2012, Thüm et al., 2014]. Dazu werden Implikationen in den Kontrakten eingeführt, sodass für ein ausgewähltes Feature seine Vor- und Nachbedingungen erfüllt werden müssen und ansonsten nicht. Wenn das Feature nicht ausgewählt ist, müssen die Vor- und Nachbedingungen aus der vorherigen Verfeinerung gelten. Bei Features, die nicht optional sind, kann die Implikation wegfallen, da die Bedingungen immer gelten müssen. Die Invarianten werden ebenfalls mit Implikation angegeben. Für die Komposition der Vor- und Nachbedingung wird Explicit Contract Refinement verwendet, da die Produktlinie *BankAccount*, die wir verifizieren wollen, damit vollständig ausgedrückt werden kann (siehe Abschnitt 4.2.6) und der Mechanismus bereits für das Metaprodukt in FeatureHouse implementiert ist [Thüm et al., 2014].

Für den Frame werden die notwendigen Felder mit Explicit Frame Refinement für die Methoden komponiert, da wir die Framing-Technik dem Explicit Contract Refinement zugeordnet haben (siehe Abschnitt 3.3). Zusätzlich muss jedoch noch die Abhängigkeit von den Features dargestellt werden. Da wir JML verwenden und dies nur Listen von Feldern gestattet [Leavens et al., 2008], können wir nicht einfach eine Implikation im Frame (in der assignable-Klausel) anwenden. Allerdings können die Abhängigkeiten von den Features in der Nachbedingung angegeben werden [Thüm et al., 2016]. Es werden also alle Felder in der assignable-Klausel belassen und wenn ein Feld x in einem optionalen Feature f verwendet wird, wird eine Nachbedingung $f ==> \backslash old(x) == x$ dem Kontrakt hinzugefügt.

Beispiel 4.2. In Quelltext 4.5 ist die Dispatcher-Methode für die Methodenverfeinerung von `update` aus dem Feature *DailyLimit* mit seiner Metaspezifikation zu sehen. Die Dispatcher-Methode führt die Methodenverfeinerung `update_DailyLimit` aus, wenn *DailyLimit* ausgewählt ist. Der Kontrakt gibt die Nachbedingungen der Originalmethode aus dem Feature *BankAccount* an, falls *DailyLimit* nicht ausgewählt

http://dx.doi.org/10.24355/dbbs.084-201711280920

ist, und die Nachbedingungen aus dem Feature *DailyLimit*, wenn *DailyLimit* ausgewählt ist. Außerdem sehen wir die Einschränkung des Frames in der Bedingung `(FM.FeatureModel.dailylimit ||\old(withdraw) == withdraw)`, falls *DailyLimit* nicht ausgewählt ist. `FM.FeatureModel` enthält die booleschen Variablen für die Featureauswahl.

Metaproduct

```
1  /*@
2    @ public normal_behavior
3    @ requires x != 0;
4    @ ensures (!(\result) || \old(balance) + x >= OVERDRAFT_LIMIT)
5    &&   (!(\old(balance) + x < OVERDRAFT_LIMIT)||!(\result))
6    &&   (!FM.FeatureModel.dailylimit||!(\result)
7        ||!(x < 0)||\old(withdraw) + x >= DAILY_LIMIT)
8    &&   (!FM.FeatureModel.dailylimit||!(\result)
9        ||\old(update_BankAccount(x)))
10   &&   (!FM.FeatureModel.dailylimit||!(x < 0)
11       ||!(\old(withdraw) + x < DAILY_LIMIT)||!(\result))
12   &&   (!FM.FeatureModel.dailylimit||\old(update_BankAccount(x))
13       ||!(\result))
14   &&   (FM.FeatureModel.dailylimit ||\old(withdraw) == withdraw);
15   @ assignable balance, withdraw;
16   @*/
17   boolean dispatch_update_DailyLimit(int x) {
18         if (FM.FeatureModel.dailylimit) {
19             return update_DailyLimit(x);
20         } else {
21             return update_BankAccount(x);
22         }
23     }
```

Quelltext 4.5: Dispatcher-Methode für die Methodenverfeinerung von der Methode `update` aus dem Feature *DailyLimit* im Metaprodukt

4.3.2 Implementierung

In diesem Abschnitt wollen wir den aktuellen Stand der Implementierung der Kontraktkompositionsmechanismen, die wir in Kapitel 3 vorgestellt haben, präsentieren. Dazu gehen wir zunächst darauf ein, wie die Mechanismen ohne Framing implementiert wurden, und anschließend welche Änderungen wir an der Implementierung vorgenommen haben. Die aktuelle Version ist in einem Fork des FeatureHouse Repositories verfügbar[2].

Einige der Kontraktkompositionsmechanismen wurden bereits so implementiert, dass das Framing entsprechend der Framing-Technik aus Kapitel 3 gehandhabt wird. Der eine Mechanismus ist Plain Contracting, bei dem einfach der Kontrakt der Originalmethode mit Frame übernommen wird. Gleiches gilt für das Contract Overriding, hier wird der Kontrakt der Methodenverfeinerung mitsamt dem Frame übernommen. Bei der Komposition im Consecutive Contract Refinement ist es ähnlich, denn dort werden die Kontrakte der Originalmethode und der Methodenverfeinerung einfach

[2]https://github.com/kruegers/featurehouse

http://dx.doi.org/10.24355/dbbs.084-201711280920

mit dem Schlüsselwort `also` verknüpft, ebenfalls inklusive des Frames.

Die anderen Kontraktkompositionsmechanismen haben keine entsprechende Implementierung für die Framing-Technik. Dementsprechend werden die Frames beim Explicit Contract Refinement, beim Conjunctive Contract Refinement und beim Cumulative Contract Refinement nicht komponiert. Für das Explicit Contract Refinement haben wir Explicit Frame Refinement vorgesehen und für Conjunctive Contract Refinement und Cumulative Contract Refinement den Frame Cut. Da der Frame Cut nicht besonders effizient ist, haben wir auch die Kombination mit Datengruppen oder Dynamic Frames diskutiert. Allerdings werden Datengruppen und Dynamic Frames ebenfalls noch nicht von FeatureHouse unterstützt.

Bei den Kontraktkompositionsmechanismen mussten wir überlegen, welche der Framing-Techniken am sinnvollsten erscheint. Dazu betrachten wir die Anwendbarkeit der Mechanismen aus dem vorangegangenen Abschnitt 4.2. Explicit Contract Refinement mit Explicit Frame Refinement deckt einen großen Teil aller Fälle ab (92%). Dagegen können Conjunctive Contract Refinement (57%) und Cumulative Contract Refinement (45%) mit Frame Cut und Datengruppen oder Dynamic Frames wesentlich weniger Kontraktkompositionen umsetzten. Außerdem ist der Aufwand für die Implementierung durch die Datengruppen und Dynamic Frames höher. Deshalb setzen wir zunächst nur das Explicit Contract Refinement um.

Für das Explicit Contract Refinement mit Explicit Frame Refinement muss im Prinzip nur das Schlüsselwort `\original` für die assignable-Klausel erweitert werden. Da hierfür die Grammatik von JML für FeatureHouse geändert werden müsste, damit `\original` in den assignable-Klauseln überhaupt verwendet werden darf, haben wir das Schlüsselwort zu `original` vereinfacht. Mit `original` in der assignable-Klausel wird jetzt also die assignable-Klausel des vorherigen Features eingebunden. Als Spezialfälle gibt es die Verwendung von `\everything` und `\nothing` zu berücksichtigen. Wenn `\everything` durch das `original` eingebunden wird, ersetzen wir die komplette assignable-Klausel durch `\everything`, da zusätzliche Felder redundant wären. Das `\nothing` wird von dem `original` nur übernommen, wenn keine anderen Felder in der assignable-Klausel angegeben worden sind, da es sonst einen Widerspruch in der Klausel geben würde.

Wir haben im letzten Abschnitt das Variability Encoding vorgestellt, das wir für die Verifikation verwenden wollen, deshalb ergänzen wir auch für das Metaprodukt eine Implementierung für das Explicit Frame Refinement. Bisher wurde das `original` implizit angenommen, das heißt die Felder aus den vorherigen Features wurden immer zur assignable-Klausel hinzugefügt. Das ganze haben wir mit dem `original` so erweitert, dass die Felder nur noch übernommen werden, wenn auch das `original` angegeben wurde.

http://dx.doi.org/10.24355/dbbs.084-201711280920

KeY-Optionen	Default-Wert	gewählter Parameter
Stop at	Default	Uncloseable
Proof splitting	Delayed	Delayed
Loop treatment	Invariant	Invariant
Block treatment	Contract	Contract
Method treatment	Contract	Contract
Dependency contracts	On	On
Query treatment	Off	On
Expand Local Queries	On	On
Arithmetic treatment	Basic	DefOps
Quantifier treatment	No split with progs	No split with progs
Class axiom rule	Free	Free
Auto Induction	Off	Off

Tabelle 4.1: Ausgewählte KeY-Optionen für die Verifikation

4.3.3 Aufbau des Experiments

Für die Verifikation verwenden wir, wie bereits erläutert, Variability Encoding. Die Metaprodukte verwenden bereits Frames, sodass wir für die Verifikation mit Frames das Metaprodukt, wie es in FeatureHouse implementiert wurde, verwenden können. Für die Verifikation ohne Frames müssen wir aus dem Metaprodukt zum einen die assignable-Klauseln entfernen und zum anderen die oben beschriebenen Nachbedingungen für den Frame. Wie bereits eingangs erläutert, wollen wir die Produktlinie *BankAccount* für die Verifikation verwenden, da sie bereits verifiziert wurde und Frames verwendet (also nicht nur `assignable \nothing`).

Die Verifikation wird mit dem Theorembeweiser KeY [Ahrendt et al., 2016] in Version 2.1 durchgeführt. Wir messen die Anzahl der Beweisschritte, die Anzahl der Beweiszweige, die Anzahl der angewendeten Regeln und die Zeit. Außerdem berücksichtigen wir, wie viele der Beweise geschlossen werden können. Für die Durchführung der Beweise verwenden wir die in Tabelle 4.1 aufgelisteten KeY-Optionen. Um die Verifikation mit Kontrakten durchzuführen sind „Block treatment Contract“, „Method treatment Contract“ und „Query treatment On“ notwendig, wobei hier nur „Query treatment“ vom Default-Wert abweicht. Außerdem müssen wir bei ein paar weiteren Parametern vom Default-Wert abweichen, die wir im Folgenden kurz erläutern. Bei „Stop at Default“ kann die Durchführung eines nicht schließbaren Beweises sehr lange dauern, aber mit „Stop at Uncloseable“ wird die Ausführung gestoppt, sobald festgestellt wird, dass der Beweis nicht schließbar ist. „Arithmetic Treatment DefOps“ wird benötigt, da in der *BankAccount* Produktlinie arithmetische Operatoren verwendet werden. Die verbliebenen Parameter können eventuell geändert werden, wodurch sich der Aufwand für die Verifikation verbessern oder verschlechtern könnte (im schlechtesten Fall können Beweise nicht geschlossen werden). Wir bleiben hier bei den Default-Werten, da es recht aufwendig wäre die optimale Kom-

http://dx.doi.org/10.24355/dbbs.084-201711280920

	alle Beweise		geschlossene Beweise	
Frames	ja	nein	ja	nein
Beweisschritte	163215	13021	16771	5718
Beweiszweige	2601	313	256	99
angewendete Regeln	296567	25933	25334	11495
Zeit in ms	192557	15038	14057	7158
geschlossene Beweise	26/26	18/26	-	-

Tabelle 4.2: Ergebnisse der Verifikation mit und ohne Frames für alle Beweise und für die Beweise, die mit und ohne Frames geschlossen wurden

bination von KeY-Optionen herauszufinden[3].

4.3.4 Auswertung der Verifikation

In Tabelle 4.2 sehen wir die Ergebnisse der Verifikation. Wir haben für den Beweisaufwand die Anzahl der Beweisschritte, der Beweiszweige und der angewendeten Regeln sowie die benötigte Zeit gemessen. Außerdem haben wir die Anzahl der geschlossenen Beweise gemessen, das heißt die Beweise konnten mit Kontrakten (mit oder ohne spezifischen Frame) bewiesen werden. In der Spalte „alle Beweise“ ist die Summe für die einzelnen Parameter für alle 26 Beweise enthalten, sowie die Anzahl der geschlossenen Beweise mit und ohne Frames. Von den Beweisen werden 18 Beweise ohne spezifische Frames geschlossen und 8 Beweise nicht. Da wir die nicht geschlossenen Beweise ohne Frames nicht mit den geschlossenen Beweisen mit Frames vom Verifikationsaufwand[4] vergleichen können, haben wir in der Spalte „geschlossene Beweise“ die einzelnen Parameter nur für die geschlossenen Beweise summiert. Interessant ist, dass die Beweise ohne spezifischen Frame weniger Beweisaufwand haben. So haben wir ohne spezifischen Frame nur 34% der Anzahl an Beweisschritten, 39% der Anzahl an Beweiszweigen, 45% der Anzahl an angewendeten Regeln und 51% der benötigten Zeit gegenüber der Verifikation mit spezifischem Frame.

In Abbildung 4.11 haben wir die benötigte Zeit für den Beweisaufwand mit und ohne spezifischen Frame für die einzelnen Methoden, die in der Verifikation mit Frames und der Verifikation ohne Frames geschlossen wurden, dargestellt. Die Verifikation mit spezifischem Frame enthält vier Methoden (`unLock`, `credit`, `estimatedInterest`, `calculateInterest`) bei denen die benötigte Zeit zwischen 11% und 33% geringer ist als ohne spezifischen Frame. Bei einer Methode (`nextYear_BankAccount`) ist die benötigte Zeit mit und ohne spezifischen Frame gleich schnell. Für die restlichen 13 Methoden ist die Verifikation ohne spezifischen Frame mit 35% (`Account`) bis 99,5% (`checkWithdraw`) der benötigten Zeit mit Frames schneller.

[3] 5 Optionen mit 3 Wahlmöglichkeiten, 1 Option mit 2 Wahlmöglichkeiten, 1 Option mit 4 Wahlmöglichkeiten → 1944 mögliche Kombinationen

[4] mit Verifikationsaufwand bezeichnen wir hier und im Folgenden die von KeY gelieferten Werte bezüglich der Anzahl der Beweisschritte, Beweiszweige und Regelanwendungen und die benötigte Zeit

http://dx.doi.org/10.24355/dbbs.084-201711280920

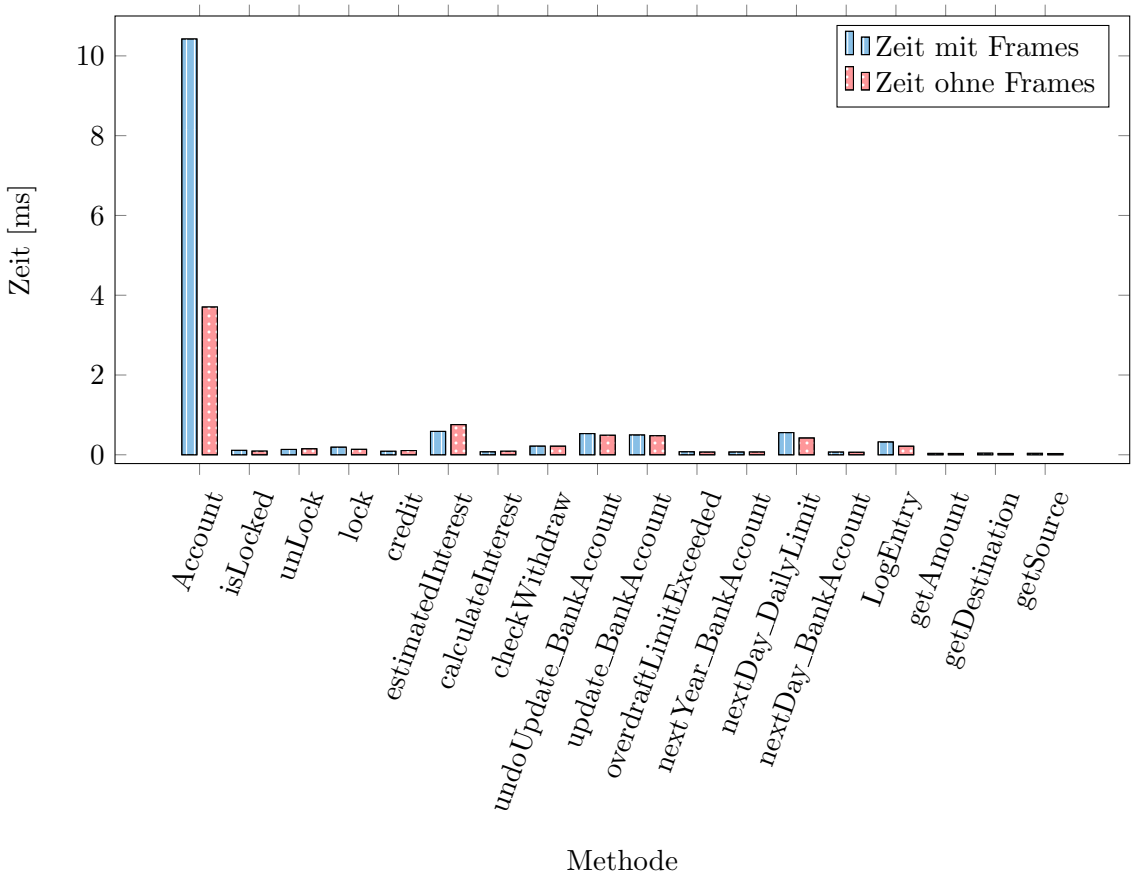

Abbildung 4.11: Benötigte Zeit für die Verifikation mit und ohne Frames

Die Anzahl der Beweisschritte ist oben in Abbildung 4.12 für die Methoden mit geschlossenem Beweis in beiden Verifikationen abgebildet. Für 15 Methoden benötigt die Verifikation ohne spezifische Frames zwischen 94% und 99,6% gegenüber der Anzahl der Beweisschritte mit Frames. Die Methoden `Account` (10%), `LogEntry` (50%) und `nextDay_DailyLimit` (81%) fallen dagegen etwas aus dem Rahmen.

In der Mitte von Abbildung 4.12 ist die Anzahl der Beweiszweige für die geschlossenen Methoden in beiden Verifikationen dargestellt. In 15 Methoden ist die Anzahl der Beweisschritte in beiden Verifikationen gleich. Die restlichen drei Methoden haben ohne Frames weniger Beweiszweige. `Account` hat 9% der Beweiszweige, `nextDay_DailyLimit` hat 76% der Beweiszweige und `LogEntry` hat 20% der Beweiszweige gegenüber der Verifikation mit Frames.

Das untere Diagramm in Abbildung 4.12 gibt die Anzahl der angewendeten Regeln für die geschlossenen Beweise der Verifikation an. Alle Methoden benötigen ohne spezifischen Frame weniger Regelanwendungen. Zwei Methoden weichen stark von den anderen ab, `Account` benötigt nur 16% und `LogEntry` nur 58% der Regelanwendungen gegenüber der Verifikation mit Frames. Die anderen 16 Methoden bewegen sich zwischen 82% und 99,5% der angewendeten Regeln gegenüber der Verifikation

http://dx.doi.org/10.24355/dbbs.084-201711280920

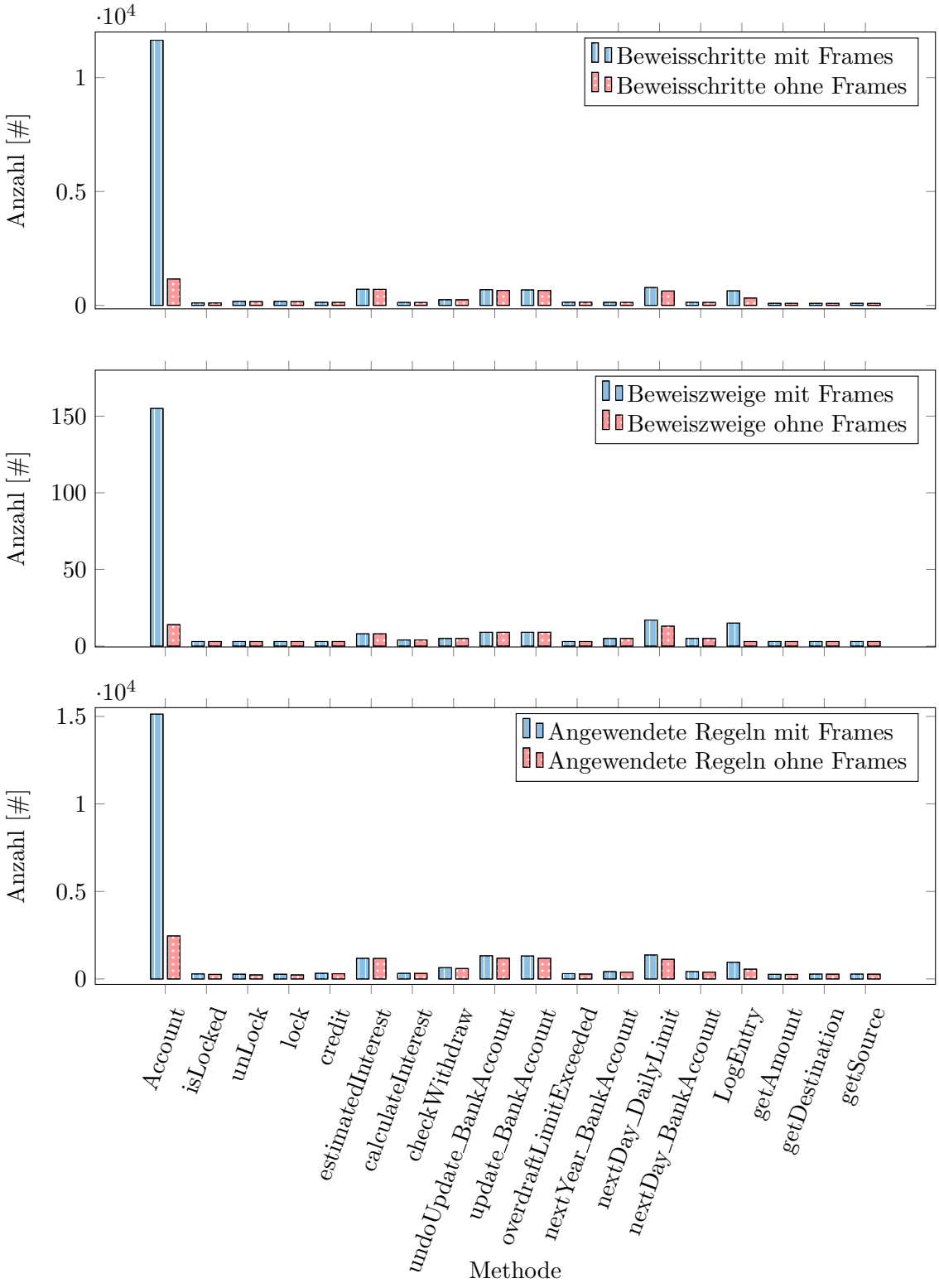

Abbildung 4.12: Benötigte Anzahl von Beweisschritten (oben), Beweiszweigen (mitte) und angewendeten Regeln (unten) mit und ohne Frames

http://dx.doi.org/10.24355/dbbs.084-201711280920

	ohne `Account`		mit `Account`	
Frames	ja	nein	ja	nein
Beweisschritte	5128	4558	16771	5718
Beweiszweige	101	85	256	99
angewendete Regeln	10206	9037	25334	11495
Zeit in ms	3633	3453	14057	7158

Tabelle 4.3: Ergebnisse der Verifikation der geschlossenen Beweise mit und ohne die Methode `Account`

mit Frames.

Der Verifikationsaufwand ist also leicht höher, wenn wir Frames verwenden. Aber wir sehen auch, dass wir ohne spezifischen Frame eher in Ausnahmefällen einen wesentlichen Vorteil gegenüber der Verifikation mit Frames haben. Insbesondere fällt bei den geschlossenen Beweisen der größte Anteil an der Anzahl der Beweisschritte (69%), der Beweiszweige (61%), der angewendeten Regeln (60%) und der benötigten Zeit (74%) auf die Methode `Account`. Für eine bessere Übersicht zeigen wir den Verifikationsaufwand für die geschlossenen Beweise noch einmal ohne `Account` in Tabelle 4.3. Ohne `Account` haben wir noch eine Differenz von 13% bei den Beweisschritten, 16% bei den Beweiszweigen, 11% bei den Regelanwendungen und 5% bei der benötigten Zeit, die Werte der Verifikation ohne Frames sind also immer noch besser. Damit können wir sagen, dass wir bei Beweisen, die sich auch ohne spezifischen Frame schließen lassen, keinen Vorteil beim Verifikationsaufwand durch die Verwendung von Frames haben, zumindest bei unserer Fallstudie *BankAccount*. Für reale Produktlinien kann sich das durchaus ändern.

Für den Verifikationsaufwand haben wir zwar keinen Vorteil, aber dafür lassen sich bei der Verifikation mit Frames gegenüber der Verifikation ohne Frames alle Beweise schließen. Deshalb wollen wir uns die acht Methoden ansehen, die mit Kontrakten ohne Frames nicht automatisch in KeY bewiesen werden konnten.

- `Account.undoUpdate_Logging`
- `Account.undoUpdate_DailyLimit`
- `Account.update_Logging`
- `Account.update_DailyLimit`
- `Application.nextYear_Interest`
- `Application.nextDay_Interest`
- `Transaction.lock`
- `Transaction.transfer`

http://dx.doi.org/10.24355/dbbs.084-201711280920

Wie wir sehen, sind nicht nur Methodenverfeinerungen betroffen, sondern auch nicht Verfeinerte. Allerdings werden in allen betroffenen Methoden Methodenaufrufe durchgeführt, die beim Beweisen mit Kontrakten mit dem dazugehörigen Kontrakt ausgewertet werden, wodurch die Beweise nicht geschlossen werden können. Einige Methoden, die ebenfalls Methodenaufrufe enthalten, können dagegen bewiesen werden. Das liegt an den Nachbedingungen des Kontrakts der aufgerufenen Methode, denn dort können auch Aussagen über den Wert eines Feldes getroffen werden, zum Beispiel durch Zuweisung eines Wertes.

Metaprodukt

```
1  /*@
2    @ public normal_behavior
3    @ requires true;
4    @ ensures true;
5    @*/
6     void nextYear_BankAccount() {
7     }
8
9  /*@
10   @ public normal_behavior
11   @ requires FM.FeatureModel.interest;
12   @ ensures account.balance == \old(account.balance)
13     + \old(account.interest)  &&  account.interest == 0;
14   @*/
15    void nextYear_Interest() {
16        nextYear_BankAccount();
17        account.balance += account.interest;
18        account.interest = 0;
19    }
```

Quelltext 4.6: Methode `nextYear_BankAccount` und `nextYear_Interest` im Metaprodukt

Beispiel 4.3. Betrachten wir in der Methode `nextYear_Interest` in Quelltext 4.6 die Nachbedingung für das Feld `account.balance` (Zeile 12+13). Die Methode ruft zuerst `nextYear_BankAccount` auf, aber die Methode gibt in der Nachbedingung des Kontrakts nicht an, was die Methode mit `account`, `account.balance` oder `account.interest` macht. Beispielsweise könnten in der Methode `nextYear_BankAccount` den Feldern `account.balance` oder `account.interest` ein neuer Wert zugewiesen werden, wodurch die Nachbedingung der Methode `nextYear_Interest` nicht mehr erfüllt wäre. Für die Verifikation mit Kontrakten, in der wir nicht auf die Methodenimplementierung von `nextYear_BankAccount` sondern nur auf den Kontrakt zugreifen, reicht der unspezifische Frame mit dem Default-Wert `assignable \everything` in der aufgerufenen Methode `nextYear_BankAccount` also nicht aus.

Wie wir in dem Beispiel sehen, hängt das Schließen des Beweises von dem Kontrakt der aufgerufenen Methode ab. Damit müssen mindestens alle Methoden, die in einer anderen Methode aufgerufen werden, mit einem Frame versehen werden, um sicherzustellen, dass wirklich alle Beweise geschlossen werden. Da aber in der Regel jede vorhandene Methode früher oder später aufgerufen wird, müssen alle Methoden mit

http://dx.doi.org/10.24355/dbbs.084-201711280920

einem möglichst kleinen Frame versehen werden, um die Produktlinie vollständig verifizieren zu können.

Abschließend können wir also sagen, dass wir zwar keinen geringeren Verifikationsaufwand haben (bezogen auf die KeY Beweisschritte, Beweiszweige, Regelanwendungen, Zeit), aber wir die Frames dennoch brauchen, damit auch definitiv alle Beweise mit Kontrakten geschlossen werden können (vorausgesetzt die Methode und der Rest des Kontraktes ist korrekt).

4.4 Threats to Validity

Die Fallstudien haben ein paar Bedrohungen für die Validität der Ergebnisse, die wir im Folgenden erläutern wollen.

Die Produktlinien sind nicht repräsentativ für reale Produktlinien [Thüm, 2015]. Das liegt daran, dass industriell verwendete Produktlinien meist größer sind beziehungsweise mehr Features besitzen, als die von uns verwendeten, aber es gibt keine großen Produktlinien mit Verträgen [Thüm, 2015]. Deshalb haben wir unter anderem die Produktlinie *GPL* verwendet, die mit 27 Features und 156 Produkten, etwas größer ist. Zudem werden Produktlinien verwendet, die auf verschiedene Arten erstellt wurden. Wie wir bereits in Abschnitt 4.1 erläutert haben, teilen sich die Produktlinien auf von Grund auf entwickelte Produktlinien mit Spezifikation, vorhandene Produktlinien, bei denen die Spezifikation ergänzt wurde, und Programme mit Spezifikation, die in Feature-Module zerlegt wurden, auf. Daher decken wir verschiedene Entwicklungsszenarien ab und reduzieren so mögliche Störfaktoren. Außerdem wurden die Produktlinien von verschiedenen Autoren entwickelt, was die Subjektivität reduziert [Thüm, 2015].

Die meisten Fallstudien wurden noch nicht verifiziert und die Kontrakte dienen eher als Dokumentation [Thüm, 2015]. Nur *BankAccount* und *StringMatcher* wurden verifiziert. Dadurch ist die Korrektheit der Kontrakte nicht gewährleistet. Eine Verifikation sollte jedoch nur zu kleinen Änderungen in den Kontrakten führen, um Fehler zu korrigieren, sie dürften also die Ergebnisse eher wenig beeinflussen. Zudem ist es unrealistisch anzunehmen, die Spezifikation sei von Beginn an fehlerfrei, da auch bei der Spezifikation Fehler vorkommen können, die erst während Tests oder der Verifikation auffallen. Dementsprechend kann angenommen werden, dass gegebenenfalls leicht fehlerhafte Kontrakte repräsentativ für reale Kontrakte in Produktlinien sind.

Die Frame-Klauseln wurden in einigen Fallstudien von uns selber ergänzt, da der Frame teilweise nicht spezifiziert war. Dadurch kann es passieren, dass der Frame nicht korrekt angegeben ist, weil die Entwickler eventuell noch andere Felder im Frame vorgesehen hätten, die aus dem Code oder der bisherigen Spezifikation nicht ersichtlich sind. Allerdings sollte der spezifizierte Frame zu den Fallstudien passen, da entweder anhand der Zuweisungen in der Implementierung oder den angegeben Nachbedingungen abgeleitet werden kann, welche Felder geändert werden müssen.

http://dx.doi.org/10.24355/dbbs.084-201711280920

Das heißt, wenn andere Felder eigentlich noch für den Frame vorgesehen wurden, sind voraussichtlich noch Fehler im Code oder der Nachbedingung vorhanden. Wie im vorherigen Absatz angegeben ist die Spezifikation nicht von vornherein fehlerfrei und da der Frame dazugehört, wird das auch für den Frame der Fall sein. Deshalb können wir davon ausgehen, dass der so angegebene Frame repräsentativ für die Produktlinien ist.

Die Daten, die in der Evaluierung verwendet wurden, sind manuell mit Hilfe von verschiedenen Tools erhoben wurden, dadurch kann es zu Fehlern in den Auszählungen kommen. Die verwendeten Tools (Kollaborationsdiagramme in FeatureIDE) wurden bereits validiert [Proksch and Krüger, 2014, Thüm, 2015], weshalb das Tooling als Fehlerquelle unwahrscheinlich ist. Die ursprünglichen Werte von Thüm [Thüm, 2015], die für uns relevant sind, haben wir überprüft. Bei den neuen Werten haben wir größtenteils nur Werte für die 60 Kontraktverfeinerungen, die wir mehrfach geprüft haben, damit auch hier Fehlerquellen ausgeschlossen werden können. Außerdem sind für die Framing-Techniken nur 19 Fälle von besonderem Interesse, die entsprechend eingehend analysiert wurden.

Eine weitere Schwierigkeit bei dem Auswerten der Fallstudien ist die Optionalität der Features. Dadurch wird es bei manchen Methodenverfeinerungen schwierig festzustellen, welche Methode verfeinert wird. Um hier Mehrdeutigkeiten zu vermeiden, haben wir festgelegt, dass ein Wert (zum Beispiel original-caller-preserving) für alle möglichen Produkte gelten muss. Das lässt sich noch gut überprüfen, da es selten vorkommt, dass ein Kontrakt mehrere Kontraktverfeinerungen hat. Das Maximum bei den Fallstudien liegt hier bei zwei Kontraktverfeinerungen, deren Features in einem Produkt ausgewählt werden können.

Wir verwenden für die Analyse des Nutzens des Framings für die Verifikation nur die Produktlinie *BankAccount*. Da wir allerdings nur aufzeigen wollten, dass ohne das Framing eine vollständige Verifikation mit Kontrakten nicht möglich ist, reicht eine Produktlinie aus.

4.5 Zusammenfassung

Wir haben zu Beginn der Arbeit angenommen, dass Framing essentiell wichtig für die Verifikation von Produktlinien ist. Durch die Verfeinerung von Methoden und ihren Kontrakten ist die Komposition der Originalframes mit den Frameverfeinerungen ebenfalls wichtig, um die Produktlinien zu verifizieren. Diese These stützen wir durch die vorangegangene Evaluierung, deren Ergebnisse wir in diesem Abschnitt nochmal diskutieren wollen.

Wir konnten anhand der Produktlinie *BankAccount* nachweisen, dass ein spezifischer Frame für die deduktive Verifikation mit Kontrakten notwendig ist (vergleiche Abschnitt 4.3). Da dort einige der Methoden (31%) ohne den Frame nicht bewiesen

http://dx.doi.org/10.24355/dbbs.084-201711280920

werden konnten, ist nachvollziehbar, wie wichtig der Frame für die Verifikation ist. Dementsprechend ist es eine berechtigte Annahme, dass wir für Feature-orientierte Programmierung Framing-Techniken benötigen.

Zunächst einmal haben wir festgestellt, dass ein paar der Produktlinien in keiner ihrer Methoden Felder ändern müssen (*StringMatcher, Numbers, Email*), also ein `assignable \nothing` vollkommen ausreichend ist, und andere Produktlinien zwar in Methoden Felder ändern, aber nicht in Methoden, die eine Kontraktverfeinerung besitzen (*UnionFind, DiGraph, IntegerSet*). Wir benötigen also in 6 Produktlinien gar keine Komposition für den Frame, während wir in 12 der Produktlinien Kontraktkompositionsmechanismen benötigen. Für größere Produktlinien sieht das vermutlich anders aus, sodass die eine oder andere Kontraktverfeinerung auch eine Frameverfeinerung besitzt.

Der Anteil der Produktlinien, in denen tatsächlich der Frame in der Kontraktverfeinerung erweitert wird, beläuft sich auf 36% der Fallstudien. Verkleinert wird der Frame in keinem Fall. Insgesamt haben wir in den 60 Kontraktverfeinerungen aus 13 Fallstudien nur 11 Frameverfeinerungen, die auf 5 Produktlinien verteilt sind. Das zeigt, dass die Komposition von Frames nur wenige Fälle betrifft, allerdings müssen die wenigen Fälle abgedeckt werden und in größeren Produktlinien werden die Fälle vermutlich auch öfter eintreten. Durch die Anwendung der vorgestellten Framing-Techniken war es möglich alle Kontraktkompositionen mit Frameverfeinerungen durchzuführen, wobei wir festgestellt haben, dass einige der Techniken überflüssig sind. So können wir alle Frameverfeinerungen, die mit Frame Overriding ausgedrückt werden können, auch mit Explicit Frame Refinement ausdrücken (das war auch schon für Contract Overriding und Explicit Contract Refinement der Fall [Thüm, 2015]). Frame Cut dagegen können wir für die Frameverfeinerungen nur anwenden, wenn wir Datengruppen oder Dynamic Frames verwenden oder Spezifikationskopien zugelassen werden (wobei das eher weniger zu empfehlen ist, da Spezifikationskopien fehleranfällig sind). Zwingend notwendig ist der Frame Cut aber auch nicht, da der Frame Cut sich eher dazu eignet Felder aus dem Frame zu entfernen als welche hinzuzufügen, was zumindest in den Fallstudien nicht vorkam.

Im Kontext der Notwendigkeit der Framing-Techniken müssen wir auch die dazugehörigen Kontraktkompositionsmechanismen berücksichtigen, denn wir haben auch festgestellt, dass bei Verfeinerung des Frames auch die Nachbedingung verfeinert wurde. Es kommt also nicht vor, dass wir die Komposition der Nachbedingung nicht brauchen, allerdings wird der Frame nie zusammen mit der Vorbedingung verfeinert. Theoretisch würden Explicit Frame Refinement und die Spezifikationsfälle ausreichen, um alle 11 Frameverfeinerungen abzudecken, allerdings haben wir bei beiden Techniken keine Preserving-Eigenschaften gegeben, weshalb es hier, soweit es möglich ist, sinnvoll ist andere Framing-Techniken zu verwenden. Plain Framing benötigen wir für die Methoden, die im Originalkontrakt einen nichtleeren Frame haben und in der Kontraktverfeinerung einen leeren Frame, um unnötigen Spezifikationsaufwand zu vermeiden, beziehungsweise in Kombination mit Plain Contracting für Methoden, die keine Kontraktverfeinerung besitzen. Frame Cut mit Datengruppen

http://dx.doi.org/10.24355/dbbs.084-201711280920

können wir verwenden, um die komponierten Kontrakte caller-preserving zu machen, weshalb auch die Technik hilfreich ist, allerdings haben wir hier gegebenenfalls wieder Spezifikationskopien.

Insgesamt lässt sich also feststellen, dass die Komposition der Frames weniger Fälle betrifft, als zu Beginn angenommen. Für die wenigen Fälle, die in den Fallstudien existieren, ist es jedoch notwendig, dass die Frames korrekt komponiert werden.

http://dx.doi.org/10.24355/dbbs.084-201711280920

http://dx.doi.org/10.24355/dbbs.084-201711280920

5. Verwandte Arbeiten

Wir haben uns in dieser Arbeit hauptsächlich mit Framing für Kontrakte in Feature-orientierter Programmierung befasst, aber es gibt auch noch andere Framing-Techniken, die wir im Folgenden näher erläutern wollen.

Datengruppen [Leino, 1998] und Dynamic Frames [Kassios, 2006] bieten eine Möglichkeit für Framing, die wir in Abschnitt 3.2 bereits erläutert haben. Mit der Adaption für Feature-orientierte Programmierung können die beiden Techniken in Zukunft auch für andere Generierungsmechanismen für Produktlinien angewendet werden.

In Abschnitt 3.1 haben wir uns mit den Feature-orientierten Kontrakten von Thüm [Thüm, 2015] ausführlich befasst. Die Feature-orientierten Kontrakte bieten Kontraktkomposition an, allerdings ohne Framing, wodurch die Kontrakte für die Verifikation noch nicht brauchbar waren, da Framing für die Verifikation zwingend notwendig ist (was wir in Abschnitt 4.3 gezeigt haben). In unserer Arbeit ergänzen wir explizit für die Kompositionsmechanismen Framing-Techniken, die es uns ermöglichen die Kontrakte für die Verifikation zu verwenden.

Variability Encoding befasst sich, wie in Abschnitt 4.3.1 besprochen, mit dem Umsetzen der Variabilität zur Laufzeit inklusive der Kontrakte mit Vor- und Nachbedingungen [Thüm et al., 2012, Thüm et al., 2014]. Die von uns präsentierten Framing-Techniken können für das Variability Encoding erweitert werden, sodass die Kontrakte für das Metaprodukt mit verschiedenen Framing-Techniken erstellt werden können.

Variability Hiding befasst sich mit der Abstraktion von Features, die von Thüm et al. [Thüm et al., 2016] mit Feature-orientierten Kontrakten ergänzt werden, sodass auch die Spezifikation abstrahiert wird. Zweck dahinter ist, die Verifikation von voneinander abhängigen Produktlinien zu erleichtern. Für die Verifikation sollen Kontrakte verwendet werden können, weshalb die Abstraktion von Kontrakten

http://dx.doi.org/10.24355/dbbs.084-201711280920

für Vor- und Nachbedingung ergänzt wird. Das Framing stellt darin jedoch noch eine Herausforderung dar, aber als potentielle Lösung werden Datengruppen angegeben, die wir in Abschnitt 3.2.2 für Feature-orientierte Kontrakte angepasst haben. Dementsprechend könnten die Datengruppen jetzt im Variability Hiding für Framing angewendet werden.

Für die Delta-orientierte Programmierung gibt es mehrere Arbeiten zur Verwendung von Kontrakten und zur Verifikation mit Kontrakten. Zunächst haben wir die Contract Deltas von Hähnle et al. [Hähnle et al., 2013], die angeben wie sich ein Kontrakt in einem Delta ändert (siehe Abschnitt 3.1.4), dazu wird, wie bei Explicit Frame Refinement, das Schlüsselwort `original` verwendet (auch für Framing), was bei den Techniken für Feature-orientierte Kontrakte dem Explicit Contract Refinement entspricht. Dem gegenüber haben wir jetzt mehrere verschiedene Kompositionsmechanismen, da nur mit Explicit Contract Refinement nicht alle Kontrakte oder auch Frames ausgedrückt werden können. Insbesondere reicht Explicit Contract Refinement nicht aus, wenn Spezifikationsfälle in den Kontrakten verwendet werden, da nicht festgelegt wird auf welchen Spezifikationsfall des Originalkontrakts sich das `original` bezieht.

Als nächstes hatten wir in Abschnitt 3.2.1 die Anwendung von Behavioral Subtyping für Delta-orientierte Programmierung von Hähnle und Schaefer [Hähnle and Schaefer, 2012], bei dem wir bereits festgestellt haben, dass es in der Form zu restriktiv in der Implementierung ist, da neu eingeführte Felder in den Deltas oder Features in Methodenverfeinerungen nicht geändert werden dürfen. Mit der Anwendung von Datengruppen oder Dynamic Frames lässt sich die Restriktion von Behavioral Subtyping lockern, die Anwendung von beidem auf die Feature-orientierte Programmierung haben wir untersucht und da Delta-orientierte Programmierung ein ähnliches Konzept hat, können Datengruppen und Dynamic Frames auch für Behavioral Subtyping in Delta-orientierter Programmierung adaptiert werden.

Eine weitere Technik ist das Delta-oriented Slicing [Bruns et al., 2011], das ebenfalls Design by Contract für die Verifikation verwendet. Das Delta-oriented Slicing stellt eine Analyse-Technik dar, die bestimmt, welche Beweise für welche Delta-Module neu durchgeführt werden müssen. Delta-oriented Slicing berücksichtigt aber kein Framing. An der Stelle könnten also gegebenenfalls die Framing-Techniken verwendet werden, um Frames zu kombinieren.

In Aspekt-orientierter Programmierung gibt es ebenfalls Ansätze für Design by Contract. Zwei Techniken, mit denen AspectJ mit JML spezifiziert werden kann, sind Pipa von Zhao und Rinard [Zhao and Rinard, 2003] oder auch AspectJML von Rebêlo et al. [Rebêlo et al., 2014a, Rebêlo et al., 2014b]. Pipa und AspectJML bieten JML Sprachkonstrukte für die Aspekte, um Invarianten und Vor- und Nachbedingungen für Advices angeben zu können. Pipa gibt zusätzlich auch noch den Frame mit assignable an, aber es fehlt noch das Einbinden der Kontrakte in den Basiscode. Dementsprechend können die in Kapitel 3 vorgestellten Framing-Techniken für Pipa

http://dx.doi.org/10.24355/dbbs.084-201711280920

und AspectJML angepasst werden, wodurch die Spezifikation wesentlich ausdrucksstärker wird.

Separation Logic ist eine von Reynolds [Reynolds, 2002] vorgestellte Erweiterung von Prädikatenlogik und Hoare Logik, die es ermöglicht über die Allokation von Speicher zu sprechen. Dafür werden zwei Kommandos für den Zugriff und das Schreiben von geteiltem Speicher hinzugefügt, separation conjunction und separation implication. Damit kann direkt in den Assertions festgelegt werden, welcher Speicher zugänglich ist. Es wird also in der Vor- oder Nachbedingung angegeben, welcher Speicher verwendet wird und nicht extra in einer Frame-Klausel. Die Notation unterscheidet sich also von der von uns verwendeten Notation in Kapitel 3, da wir direkt über eine Frame-Klausel sprechen und den Frame nicht in Vor- und Nachbedingung ausdrücken. Wir verwenden Separation Logic deshalb nicht, da auch einige Spezifikationssprachen (zum Beispiel JML assignable, SPEC# modifies) eine Notation mit Frame-Klausel verwenden und insbesondere die Kontraktkompositionsmechanismen mit JML bereits umgesetzt worden sind [Thüm, 2015] und wir die Arbeit darauf aufgebaut haben. Separation Logic gibt es noch nicht für Produktlinien, weshalb es eine Überlegung wert wäre, die Framing-Techniken für die Umsetzung von Seperation Logic in Produktlinien zu verwenden.

http://dx.doi.org/10.24355/dbbs.084-201711280920

http://dx.doi.org/10.24355/dbbs.084-201711280920

6. Zusammenfassung

Software-Produktlinien ermöglichen eine einfache Entwicklung einer Vielzahl von Produkten anhand einer gemeinsamen Codebasis und variablen Merkmalen. Insbesondere durch den Einsatz in sicherheitskritischen Systemen wird die Verifikation notwendig, die sich jedoch durch die mitunter enorme Produktvielfalt als besondere Herausforderung darstellt.

In dieser Arbeit haben wir uns mit der deduktiven Verifikation mit Feature-orientierten Kontrakten befasst. Die Kontraktkompositionsmechanismen der Feature-orientierten Kontrakte decken bisher allerdings nur die Vor- und Nachbedingung der Kontrakte ab, die Komposition der Frames wurde noch nicht diskutiert. Deshalb haben wir das Framing für die Kontraktkomposition in Feature-orientierter Programmierung untersucht. Aus den Eigenschaften der Kontraktkompositionsmechanismen haben wir fünf Framing-Techniken abgeleitet, die die Komposition der Frames für Feature-orientierte Kontrakte beschreiben. Da die Framing-Techniken aber teilweise Einschränkungen in der Implementierung aufweisen, verwenden wir Techniken aus der Objekt-orientierten Programmierung (Datengruppen, Dynamic Frames), um die Framing-Techniken zu verbessern.

Anschließend haben wir die entwickelten Framing-Techniken anhand von 14 Fallstudien evaluiert. Wir haben festgestellt, dass die Methoden häufig keine Felder modifizieren (45% der Kontrakte mit `assignable \nothing`) und Kontraktverfeinerungen selten Modifikationen an Frames vornehmen (11 von 60 Kontraktverfeinerungen). Wir haben nur kleine Produktlinien verwendet, weshalb es wahrscheinlich ist, dass die Anzahl für reale Produktlinien größer ist. Die Analyse der Anwendbarkeit der Framing-Technik hat ergeben, dass keine Technik alle Fälle abdecken kann, aber wir mit mehreren Techniken eine vollständige Spezifikation ermöglichen können.

Mit der Verifikation von einer Produktlinie mit und ohne spezifischen Frame geben wir einen Einblick darüber, welche Notwendigkeit die Frames für die Verifikation mit

http://dx.doi.org/10.24355/dbbs.084-201711280920

Kontrakten haben. Bei der Verifikation haben wir festgestellt, dass die Verwendung von Kontrakten in der Verifikation nur möglich ist, wenn repräsentative Frames für die Methoden vorhanden sind, also nur Felder angegeben werden, die tatsächlich in der Methode geändert werden müssen (im Gegensatz zu `assignable \everything`). Ansonsten kann es passieren, dass sich einige Beweise nicht schließen lassen (in unserer Fallstudie waren 8 von 26 Beweisen betroffen).

Mithilfe der Frames können wir jetzt die Feature-orientierten Kontrakte für die Verifikation einsetzen. Ohne die Frames war die Spezifikation durch die Kontrakte für eine Verifikation nicht ausreichend, wodurch es wichtig war sich mit der Komposition der Frames in Feature-orientierten Kontrakten zu befassen. Selbst bei einer kleinen Produktlinie, wie *BankAccount*, konnte so keine vollständige Verifikation mit Kontrakten durchgeführt werden, dementsprechend wird es bei einer realen Produktlinie nicht besser werden. Die entwickelten Framing-Techniken decken verschiedene Kompositionen ab, wodurch es in den Fallstudien möglich war, alle Kontraktkompositionen auszudrücken und es auch möglich sein sollte in größeren Produktlinien alle Frames in den Features zu spezifizieren.

http://dx.doi.org/10.24355/dbbs.084-201711280920

7. Zukünftige Arbeiten

In dieser Arbeit haben wir uns mit der Erweiterung von Kontraktkompositionsmechanismen für Feature-orientierte Programmierung befasst (vergleiche Kapitel 3) und der Evaluierung in 14 Fallstudien (siehe Kapitel 4), aber wir haben noch nicht alle Aspekte berücksichtigt. In diesem Kapitel diskutieren wir welche Aspekte bezüglich der Kontraktkomposition in Produktlinien noch betrachtet werden können.

Ein Punkt im Framing ist noch die Abhängigkeit von Variablen, was in JML mit der accessible-Klausel ausgedrückt wird [Leavens et al., 2008]. In der accessible-Klausel wird damit angegeben, welche Felder während der Ausführung einer Methode gelesen werden dürfen. Hier können gegebenenfalls die Techniken, die wir für die assignable-Klausel verwenden, übertragen werden. Die Frage hier ist jedoch, wie das accessible die Verifikation beeinflusst. Insbesondere, ob die accessible-Klausel mit dem Default `\everything` gegenüber einer spezifischeren accessible-Klausel Einfluss auf die Schließbarkeit oder die Größe der Beweise hat.

Weiterhin haben wir in Kapitel 4 festgestellt, dass Spezifikationsfälle kritisch sind im Hinblick auf Kontraktkompositionen und sich meistens nur mit Consecutive Contract Refinement ausdrücken lassen. Bei der Implementierung der Kontrakte für Vor- und Nachbedingungen gab es ähnliche Schwierigkeiten [Benduhn, 2012]. Für Explicit Contract Refinement wurden als Lösung drei verschiedene Schlüsselwörter vorgestellt, eins um die ganze Spezifikation zu referenzieren, eins um einen Spezifikationsfall zu referenzieren und eins, das alle Spezifikationsbedingungen eines Typs referenziert (Vorbedingung, Nachbedingung). Eine Erweiterung auf Framing würde das Problem, das wir in Beispiel 4.1 hatten, lösen. Dort kann es jedoch auch zu Schwierigkeiten kommen, zum Beispiel wenn wir Kontrakte in verschiedenen Features verfeinern und die Kontrakte eine unterschiedliche Anzahl von Spezifikationsfällen aufweisen oder die Spezifikationsfälle nicht zueinander gehören. Außerdem ist damit noch nicht geklärt, wie Spezifikationsfälle in Conjunctive und Cumulative Contract Refinement erweitert werden können.

http://dx.doi.org/10.24355/dbbs.084-201711280920

Ebenfalls wäre eine Übertragung der Kontraktkomposition mit Framing auf andere Techniken für die Entwicklung von Produktlinien, wie zum Beispiel Delta-orientierte Programmierung oder Aspekt-orientierte Programmierung, denkbar. Viele Techniken haben noch keine oder keine hinreichende Unterstützung von Kontrakten und insbesondere nicht mit Framing (vergleiche Kapitel 1). Nur Delta-orientierte Programmierung unterstützt bereits Kontraktverfeinerungen mit Framing [Hähnle et al., 2013]. Aber wie bereits für die Kontraktkompositionen ohne Framing für Feature-orientierte Programmierung festgestellt wurde [Thüm, 2015], reicht eine Kompositionstechnik nicht aus, um alle Kontraktkompositionen darstellen zu können.

Der vollständige Tool-Support für die Framing-Techniken steht noch aus. Die Kontraktkompositionsmechanismen [Thüm, 2015] und ein paar der Framing-Techniken wurde bereits in FeatureIDE mit FeatureHouse umgesetzt. Plain Contracting, Contract Overriding und Consecutive Contract Refinement standen bereits mit Framing-Techniken zur Verfügung. Wir haben Explicit Frame Refinement im Explicit Contract Refinement ergänzt (siehe Abschnitt 4.3.2). Dementsprechend fehlt noch die Umsetzung vom Frame Cut für Conjunctive Contract Refinement und Cumulative Contract Refinement für den wir zusätzlich noch die Datengruppen oder die Dynamic Frames benötigen. Datengruppen sind im JML Standard enthalten [Leavens et al., 2008] und Dynamic Frames werden in der von Weiß vorgestellten Erweiterung JML* [Weiß, 2011] ergänzt. Da wir eher die Verifikation von Produktlinien als Ziel betrachten, als nur die Dokumentation, schlagen wir die Verwendung von Dynamic Frames vor, da diese von Tools, zum Beispiel KeY, unterstützt werden und damit die Verifikation durchgeführt werden kann, da Datengruppen Probleme mit Local Reasoning bereiten [Weiß, 2011].

Anschließend kann eine umfassende Evaluierung bezüglich des Nutzens von Framing in der Verifikation von Produktlinien durchgeführt werden, da wir bisher nur die Verifikation mit einem Metaprodukt berücksichtigt haben und die Kontraktkompositionsmechanismen dafür nicht verwenden. Dementsprechend wäre es sinnvoll noch zu analysieren, wie hoch der Nutzen der Framing-Techniken in der Verifikation ist. Dafür können beispielsweise für eine oder mehrere Produktlinien alle Produkte generiert werden und anschließend ohne Frame und mit den verschiedenen Framing-Techniken verifiziert werden. Da wir geplant haben für jeden Kontraktkompositionsmechanismus nur eine Framing-Technik zu benutzen, reicht es hier den Nutzen abhängig von den verschiedenen Mechanismen mit Framing zu bestimmen. Wahlweise kann auch der Nutzen für die einzelnen Produktlinien bestimmt werden, indem nur der bestmögliche Kompositionsmechanismus für jede Methode gewählt wird, da es ohnehin nicht unbedingt möglich ist mit einem Mechanismus alle Kontraktkompositionen zu beschreiben.

http://dx.doi.org/10.24355/dbbs.084-201711280920

A. Anhang

Wir haben verschiedene Produktlinien für die Evaluierung verwendet, die wir kurz präsentieren wollen. Die Produktlinien wurden bereits 2014 für die Evaluierung von Feature-orientierten Kontrakten ohne Framing von Thomas Thüm verwendet [Thüm, 2015], dort findet sich ebenfalls eine Beschreibung aller verwendeter Produktlinien. Außerdem stellen wir noch Tabellen mit den Eigenschaften, die wir in der Evaluierung untersucht haben bereit. Dazu haben wir in Tabelle A.1 Statistiken zu dem Aufbau der Implementierung der Produktlinien angelegt, unter anderem die Anzahl der Methoden und Methodenverfeinerungen. In Tabelle A.2 sind Informationen zu den Kontrakten der Produktlinien zu finden, das heißt zum Beispiel Anzahl von Kontrakten, nichtleeren Frames und Frameverfeinerungen. Zuletzt sind in Tabelle A.3 die Eigenschaften der Kontraktverfeinerungen zu sehen (Preserving-Eigenschaften, welcher Teil des Kontrakts verfeinert wird).

Die aktuellen Versionen der Produktlinien sind im FeatureIDE Repository[1] zu finden. Von der Produktlinie *BankAccount* haben wir die Version von Krüger [Krüger, 2014] für die Verifikation mit KeY verwendet.

Übersicht über die Produktlinien

Wir haben zunächst die Produktlinien, die von Grund auf mit Features und Kontrakten entwickelt worden sind:

- *BankAccount*: Es handelt sich hierbei um eine Produktlinie, die Basisfunktionen für eine Bank liefert mit Bankkonten und verschiedenen Erweiterungen, wie zum Beispiel Transaktionen oder ein tägliches Abhebungslimit. Die *BankAccount* Produktlinie wurde von Thomas Thüm 2011 entwickelt und mit COQ verifiziert [Thüm et al., 2011], 2012 um das Feature *Overdraft* erweitert und

[1] https://github.com/FeatureIDE/FeatureIDE/tree/develop/plugins/de.ovgu.featureide.examples/featureide_examples

http://dx.doi.org/10.24355/dbbs.084-201711280920

	Produkte	Features	Klassen	Felder	Methoden	Methodenverfeinerungen
BankAccount	72	8	3	8	13	5
GPL-scratch	128	10	4	13	42	7
IntegerList	16	5	2	2	9	4
UnionFind	6	8	4	8	19	10
StringMatcher	6	8	2	0	3	6
DiGraph	8	4	8	12	66	0
ExamDB	8	8	4	10	50	29
IntegerSet	2	3	1	7	15	6
Numbers	2	2	1	0	10	7
Paycard	6	4	7	42	25	3
Poker	21	10	8	34	56	5
Elevator	20	6	11	38	92	19
Email	40	9	3	23	55	16
GPL	156	27	21	46	104	53

Tabelle A.1: Statistiken der Produktlinien

mit KeY verifiziert [Thüm et al., 2012] und 2013 wurden von Jens Meinicke zwei weitere Features ergänzt und mit KeY und dem Java Path Finder (JPF) verifiziert [Meinicke, 2013, Thüm et al., 2014]. In der Version von Stefan Krüger wurden Änderungen vorgenommen [Krüger, 2014], um das Metaprodukt (siehe Abschnitt 4.3.1) zu beweisen, da wir ebenfalls das Metaprodukt für die Verifikation verwenden, benutzen wir diese Version, allerdings ohne das Feature *TransactionLog*.

- *GPL-scratch*: *GPL-scratch* liefert eine umfassende Implementation für Graphen, deren Aufbau und verschiedenen Algorithmen, zum Beispiel Breiten- und Tiefensuche. Die Produktlinie wurde aus der Produktlinie *GPL* entwickelt indem sie in Feature-Module zerlegt wurde. Die Produktlinie wurde von André Weigelt für die Evaluierung der Feature-orientierten Kontrakte mit FeatureIDE entwickelt [Weigelt, 2013] und basiert auf dem Standardproblem für Produktlinien von Lopez-Herrejon und Batory [Lopez-Herrejon and Batory, 2001]. Die Frames haben wir für die Evaluierung in der Produktlinie ergänzt.

- *IntegerList*: *IntegerList* ist eine Implementierung von einer Listen Datenstruktur, die Integer speichert, mit Basisoperatoren, wie Sortieren. Die Produktlinie wurde für die Evaluierung von der Entdeckung von Feature-Interaktionen mithilfe von Kontrakten von Wolfgang Scholz et al. entwickelt [Scholz et al., 2011].

http://dx.doi.org/10.24355/dbbs.084-201711280920

	Kontrakte	Kontraktverfeinerungen	Frames	Frameverfeinerungen	nichtleere Frames	Explict Frame Refinement	Plain Framing	Frame Overriding	Frame Cut	Consecutive Frame Refinement	Datengruppen	Dynamic Frames
BankAccount	18	9	5	5	5	5	0	2	0	5	4	4
GPL-scratch	42	8	7	0	1	1	1	0	0	0	1	1
IntegerList	7	4	1	0	1	1	1	0	0	1	1	1
UnionFind	7	2	2	0	0	0	0	0	0	0	0	0
StringMatcher	7	0	6	0	0	0	0	0	0	0	0	0
DiGraph	45	21	0	0	0	0	0	0	0	0	0	0
ExamDB	40	8	15	0	1	0	0	0	0	1	1	1
IntegerSet	12	9	0	0	0	0	0	0	0	0	0	0
Numbers	15	0	6	0	0	0	0	0	0	0	0	0
Paycard	10	4	1	1	1	0	0	0	0	1	1	1
Poker	48	34	2	1	2	2	1	0	0	0	2	2
Elevator	13	6	6	1	5	5	4	0	0	5	5	5
Email	7	0	3	0	0	0	0	0	0	0	0	0
GPL	110	66	6	3	3	3	0	0	0	3	3	3

Tabelle A.2: Statistiken der Kontrakte in den Produktlinien

Die vorhandenen Kontrakte haben wir im Rahmen der Arbeit mit Frames ergänzt.

- *UnionFind*: Implementierung von Fabian Benduhn [Benduhn, 2012] des Union-Find-Algorithmus [Sedgewick and Wayne, 1983] in verschiedenen Varianten. Die Kontrakte wurden von uns mit Frames erweitert.
- *StringMatcher*: *StringMatcher* ist eine Produktlinie, die verschiedene Operationen zum Vergleichen von Strings bietet. Die Produktlinie wurde ebenfalls von Fabian Benduhn entwickelt [Thüm, 2015] und verifiziert [Thüm et al., 2014]. Den Frame in den Kontrakten haben wir mit `assignable \nothing` in den Kontrakten ergänzt, da hier keine Felder vorhanden sind.

Als nächstes haben wir die bestehenden spezifizierten Programme, die in eine Produktlinie zerlegt worden sind:

- *DiGraph*: *DiGraph* modelliert einen gerichteten Graphen inklusive einiger Operationen, wie Transposition oder Suche. Ursprünglich war *DiGraph* ein Beispiel für JML von Albert Baker, Katie Becker und Gary T. Leavens[2], die von Tho-

[2]http://www.eecs.ucf.edu/ leavens/JML-release/org/jmlspecs/samples/digraph/

http://dx.doi.org/10.24355/dbbs.084-201711280920

	original-caller-preserving	original-callee-preserving	refinement-caller-preserving	refinement-callee-preserving	original-preserving	refinement-preserving	caller-preserving	callee-preserving	Vor- und Nachbedingung	Vorbedingung	Nachbedingung	Identische Kontrakte	Nachbedingung und Frame
BankAccount	0	0	2	2	0	2	0	0	0	0	0	0	5
GPL-scratch	2	2	2	2	0	0	2	2	6	1	0	0	0
IntegerList	1	0	0	0	0	0	0	0	0	0	1	0	0
UnionFind	2	0	0	2	0	0	0	0	0	0	2	0	0
StringMatcher	6	0	2	4	0	0	2	0	0	0	6	0	0
DiGraph	0	0	0	0	0	0	0	0	0	0	0	0	0
ExamDB	3	0	3	0	0	0	3	0	0	1	14	0	0
IntegerSet	0	0	0	0	0	0	0	0	0	0	0	0	0
Numbers	6	0	6	0	0	0	6	0	0	0	6	0	0
Paycard	0	0	0	0	0	0	0	0	0	0	0	0	1
Poker	0	1	0	1	0	0	0	1	0	1	0	0	1
Elevator	5	0	1	0	0	0	1	0	0	0	5	0	1
Email	1	2	0	1	0	0	0	1	0	2	1	0	0
GPL	2	3	3	3	2	3	2	3	0	0	1	2	3

Tabelle A.3: Eigenschaften der Kontraktverfeinerungen mit Framing in den Produktlinien

mas Thüm in eine Produktlinie mit dem Basis Feature *DiGraph* und drei Features *Transposeable*, *Searchable* und *Removal* zerlegt wurde [Thüm, 2015].

- *ExamDB*: Timo Eiffler hat ein System für das Klausurmanagement implementiert[3] und mit KeY verifiziert. Thomas Thüm hat *ExamDB* mit und ohne Pure-Method-Refinements modifiziert und in eine Produktlinie zerlegt [Thüm, 2015]. Wir verwenden die Version ohne Pure-Method-Refinements.

- *IntegerSet*: *IntegerSet* modelliert Mengen für Integer mit verschiedenen Datenstrukturen (Bäume, Hashsets). Das Programm wurde ebenfalls als JML-Beispiel von Katie Becker, Arthur Thomas und Gary T. Leavens implementiert[4] und von Fabian Benduhn in eine Produktlinie zerlegt [Benduhn, 2012].

- *Numbers*: *Numbers* bietet Basisoperationen für wahlweise reelle oder komplexe Zahlen, die Fabian Benduhn in eine Produktlinie gebracht hat [Benduhn, 2012]. Der ursprüngliche Quellcode stammt aus einem JML-Beispiel von Gary T. Leavens[5].

[3] Veröffentlicht im VerifyThis Repository [Thüm, 2015]
[4] http://www.eecs.ucf.edu/ leavens/JML-release/org/jmlspecs/samples/sets/
[5] http://www.eecs.ucf.edu/ leavens/JML-release/org/jmlspecs/samples/dbc/

http://dx.doi.org/10.24355/dbbs.084-201711280920

- *Paycard*: *Paycard* ist eine Implementierung für Zahlungskarten, die aufgeladen und zur Zahlung benutzt werden können. Der Code stammt ursprünglich aus einem Beispiel für KeY[6] und wurde von Thomas Thüm in eine Produktlinie umgewandelt [Thüm, 2015]. Wir haben die fehlenden Frames in den Kontrakten für die Evaluierung ergänzt.

- *Poker*: *Poker* ist eine Implementierung für ein Pokerspiel[7]. Fabian Benduhn hat den Code in eine Produktlinie mit verschiedenen Features umgewandelt [Benduhn, 2012], zum Beispiel mit Wetten und maximaler Kartenanzahl pro Spieler.

Zuletzt haben wir noch die existierenden Produktlinien, die im Nachhinein mit Kontrakten versehen wurden:

- *Elevator*: Ein Aufzugsystem von Plath und Ryan [Plath and Ryan, 2001], das von Alexander von Rhein mit AspectJ[8] implementiert wurde [Apel et al., 2013d], mit dem das Verhalten eines Aufzugs mit optionalen Features, wie Maximalgewicht oder Priorisierung von Etagen, beschrieben wird. Thomas Thüm hat 2013 die AspectJ Spezifikation in Feature-orientierte Kontrakte transferiert [Thüm, 2015]. Im Rahmen der Arbeit haben wir den Frame in der Spezifikation ergänzt.

- *Email*: Ein Emailsystem von Hall [Hall, 2005], das verschiedene Optionen, zum Beispiel Verschlüsselung bietet. Alexander von Rhein, Stefan Boxleitner und Hendrik Speidel haben das System in Java mit AspectJ implementiert[9] [Apel et al., 2013d] und Thomas Thüm hat die AspectJ Spezifikation in Feature-orientierten Kontrakten umgesetzt [Thüm, 2015]. Für die Evaluierung haben wir den Frame in den vorhandenen Kontrakten ergänzt, die wir aber alle als `assignable \nothing` angegeben haben.

- *GPL*: Wie oben bereits beschrieben eine Produktlinie für die Modellierung von Graphen [Lopez-Herrejon and Batory, 2001]. Die Spezifikation wurde 2012 von Fabian Benduhn ergänzt [Thüm, 2015]. Hier haben wir ebenfalls die Kontrakte mit einem Frame erweitert.

[6] http://i12www.ira.uka.de/key/download/quicktour/
[7] Ursprungscode: https://github.com/topless/PokerTop
[8] http://spl2go.cs.ovgu.de/projects/16
[9] http://spl2go.cs.ovgu.de/projects/17

http://dx.doi.org/10.24355/dbbs.084-201711280920

http://dx.doi.org/10.24355/dbbs.084-201711280920

Literaturverzeichnis

[Ahrendt et al., 2016] Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P. H., and Ulbrich, M. (2016). *Deductive Software Verification–The KeY Book: From Theory to Practice*, volume 10001 of LNCS. Springer. (zitiert auf Seite 2, 12, 36 und 64)

[Apel et al., 2013a] Apel, S., Batory, D., Kästner, C., and Saake, G. (2013a). *Feature-Oriented Software Product Lines*. Springer. (zitiert auf Seite 1 und 5)

[Apel et al., 2013b] Apel, S., Kästner, C., and Lengauer, C. (2013b). Language-Independent and Automated Software Composition: The FeatureHouse Experience. *IEEE Transactions on Software Engineering*, 39(1):63–79. (zitiert auf Seite 1, 7 und 30)

[Apel et al., 2010] Apel, S., Lengauer, C., Möller, B., and Kästner, C. (2010). An algebraic foundation for automatic feature-based program synthesis. *Science of Computer Programming*, 75(11):1022 – 1047. Special Section on the Programming Languages Track at the 23rd ACM Symposium on Applied Computing. (zitiert auf Seite 7)

[Apel et al., 2013c] Apel, S., Rhein, A. v., Wendler, P., Größlinger, A., and Beyer, D. (2013c). Strategies for product-line verification: Case studies and experiments. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 482–491, Piscataway, NJ, USA. IEEE Press. (zitiert auf Seite 61)

[Apel et al., 2013d] Apel, S., Rhein, A. v., Wendler, P., Grösslinger, A., and Beyer, D. (2013d). Strategies for product-line verification: Case studies and experiments. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 482–491, Piscataway, NJ, USA. IEEE Press. (zitiert auf Seite 87)

[Barnes, 1997] Barnes, J. (1997). *High integrity Ada: the SPARK approach*. Addison-Wesley Professional. (zitiert auf Seite 12)

[Barnett et al., 2005] Barnett, M., Leino, K. R. M., and Schulte, W. (2005). *The Spec# Programming System: An Overview*, pages 49–69. Springer Berlin Heidelberg, Berlin, Heidelberg. (zitiert auf Seite 12)

[Batory, 2005] Batory, D. (2005). *Feature Models, Grammars, and Propositional Formulas*, pages 7–20. Springer Berlin Heidelberg, Berlin, Heidelberg. (zitiert auf Seite 6)

http://dx.doi.org/10.24355/dbbs.084-201711280920

[Batory et al., 2004] Batory, D., Sarvela, J. N., and Rauschmayer, A. (2004). Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371. (zitiert auf Seite 1 und 7)

[Beckert and Hähnle, 2014] Beckert, B. and Hähnle, R. (2014). Reasoning and verification: State of the art and current trends. *IEEE Intelligent Systems*, 29(1):20–29. (zitiert auf Seite 1, 2 und 60)

[Benduhn, 2012] Benduhn, F. (2012). Contract-aware feature composition. *Bachelor's thesis, University of Magdeburg, Germany.* (zitiert auf Seite 56, 81, 85, 86 und 87)

[Bruns et al., 2011] Bruns, D., Klebanov, V., and Schaefer, I. (2011). *Verification of Software Product Lines with Delta-Oriented Slicing*, pages 61–75. Springer Berlin Heidelberg, Berlin, Heidelberg. (zitiert auf Seite 2 und 76)

[Clements and Northrop, 2001] Clements, P. and Northrop, L. (2001). *Software Product Lines: Practices and Patterns*. Addison-Wesley Professional. (zitiert auf Seite 1, 5 und 6)

[Damiani et al., 2012] Damiani, F., Owe, O., Dovland, J., Schaefer, I., Johnsen, E. B., and Yu, I. C. (2012). A transformational proof system for delta-oriented programming. In *Proceedings of the 16th International Software Product Line Conference - Volume 2*, SPLC '12, pages 53–60, New York, NY, USA. ACM. (zitiert auf Seite 2)

[Dhara and Leavens, 1996] Dhara, K. K. and Leavens, G. T. (1996). Forcing behavioral subtyping through specification inheritance. In *Proceedings of the 18th International Conference on Software Engineering*, ICSE '96, pages 258–267, Washington, DC, USA. IEEE Computer Society. (zitiert auf Seite 26)

[Hähnle and Schaefer, 2012] Hähnle, R. and Schaefer, I. (2012). *A Liskov Principle for Delta-Oriented Programming*, pages 32–46. Springer Berlin Heidelberg, Berlin, Heidelberg. (zitiert auf Seite 2, 32 und 76)

[Hähnle et al., 2013] Hähnle, R., Schaefer, I., and Bubel, R. (2013). *Reuse in Software Verification by Abstract Method Calls*, pages 300–314. Springer Berlin Heidelberg, Berlin, Heidelberg. (zitiert auf Seite 2, 23, 76 und 82)

[Hall, 2005] Hall, R. J. (2005). Fundamental nonmodularity in electronic mail. *Automated Software Engineering*, 12(1):41–79. (zitiert auf Seite 87)

[Hatcliff et al., 2012] Hatcliff, J., Leavens, G. T., Leino, K. R. M., Müller, P., and Parkinson, M. (2012). Behavioral interface specification languages. *ACM Comput. Surv.*, 44(3):16:1–16:58. (zitiert auf Seite 2, 11, 12, 30 und 31)

[Hoare, 1969] Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580. (zitiert auf Seite 17)

http://dx.doi.org/10.24355/dbbs.084-201711280920

[Kang et al., 1990] Kang, K., Cohen, S., Hess, J., Novak, W., and Peterson, A. (1990). Feature-oriented domain analysis (foda) feasibility study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA. (zitiert auf Seite 6)

[Kassios, 2006] Kassios, I. T. (2006). *Dynamic Frames: Support for Framing, Dependencies and Sharing Without Restrictions*, pages 268–283. Springer Berlin Heidelberg, Berlin, Heidelberg. (zitiert auf Seite 2, 37 und 75)

[Kiczales et al., 1997] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. (1997). *Aspect-oriented programming*, pages 220–242. Springer Berlin Heidelberg, Berlin, Heidelberg. (zitiert auf Seite 10)

[Klaus Pohl and van der Linden, 2005] Klaus Pohl, G. B. and van der Linden, F. J. (2005). *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag Berlin Heidelberg. (zitiert auf Seite 5 und 6)

[Krüger, 2014] Krüger, S. (2014). Product-line verifcation with abstract contracts. Master's thesis, Magdeburg, Universität, 2014. (zitiert auf Seite 6, 16, 83 und 84)

[Leavens, 1996] Leavens, G. T. (1996). *An Overview of Larch/C++: Behavioral Specifications for C++ Modules*, pages 121–142. Springer US, Boston, MA. (zitiert auf Seite 12)

[Leavens et al., 2006] Leavens, G. T., Baker, A. L., and Ruby, C. (2006). Preliminary Design of JML: A Behavioral Interface Specification Language for Java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38. (zitiert auf Seite 12 und 33)

[Leavens and Müller, 2007] Leavens, G. T. and Müller, P. (2007). Information hiding and visibility in interface specifications. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 385–395, Washington, DC, USA. IEEE Computer Society. (zitiert auf Seite 12)

[Leavens et al., 2008] Leavens, G. T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., Kiniry, J., Chalin, P., Zimmerman, D. M., et al. (2008). JML reference manual. (zitiert auf Seite 27, 61, 81 und 82)

[Leino, 1998] Leino, K. R. M. (1998). Data groups: Specifying the modification of extended state. In *ACM SIGPLAN Notices*, volume 33, pages 144–153. ACM. (zitiert auf Seite 33 und 75)

[Leino and Nelson, 2002] Leino, K. R. M. and Nelson, G. (2002). Data abstraction and information hiding. *ACM Trans. Program. Lang. Syst.*, 24(5):491–553. (zitiert auf Seite 2, 12 und 36)

[Liskov and Wing, 1994] Liskov, B. H. and Wing, J. M. (1994). A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841. (zitiert auf Seite 30)

[Lopez-Herrejon and Batory, 2001] Lopez-Herrejon, R. E. and Batory, D. S. (2001). A standard problem for evaluating product-line methodologies. In *Proceedings of*

http://dx.doi.org/10.24355/dbbs.084-201711280920

the Third International Conference on Generative and Component-Based Software Engineering, GCSE '01, pages 10–24, London, UK, UK. Springer-Verlag. (zitiert auf Seite 84 und 87)

[Meinicke, 2013] Meinicke, J. (2013). JML-based verification for feature-oriented programming. *Bachelorarbeit, University of Magdeburg, Germany*, pages 2–21. (zitiert auf Seite 84)

[Meyer, 1988] Meyer, B. (1988). *Object-oriented software construction*, volume 2. Prentice hall New York. (zitiert auf Seite 25)

[Meyer, 1992] Meyer, B. (1992). Applying 'design by contract'. *Computer*, 25(10):40–51. (zitiert auf Seite 2 und 12)

[Plath and Ryan, 2001] Plath, M. and Ryan, M. (2001). Feature integration using a feature construct. *Science of Computer Programming*, 41(1):53–84. (zitiert auf Seite 87)

[Prehofer, 1997] Prehofer, C. (1997). *Feature-oriented programming: A fresh look at objects*, pages 419–443. Springer Berlin Heidelberg, Berlin, Heidelberg. (zitiert auf Seite 7 und 30)

[Proksch and Krüger, 2014] Proksch, F. and Krüger, S. (2014). Tool Support for Contracts in FeatureIDE. Technical report, Technical Report FIN-001-2014, University of Magdeburg, Germany. (zitiert auf Seite 71)

[Rebêlo et al., 2014a] Rebêlo, H., Leavens, G. T., Bagherzadeh, M., Rajan, H., Lima, R., Zimmerman, D. M., Cornélio, M., and Thüm, T. (2014a). AspectJML: Modular Specification and Runtime Checking for Crosscutting Contracts. In *Proceedings of the 13th International Conference on Modularity*, MODULARITY '14, pages 157–168, New York, NY, USA. ACM. (zitiert auf Seite 2 und 76)

[Rebêlo et al., 2014b] Rebêlo, H., Leavens, G. T., Bagherzadeh, M., Rajan, H., Lima, R., Zimmerman, D. M., Cornélio, M., and Thüm, T. (2014b). Modularizing Crosscutting Contracts with AspectJML. In *Proceedings of the Companion Publication of the 13th International Conference on Modularity*, MODULARITY '14, pages 21–24, New York, NY, USA. ACM. (zitiert auf Seite 2 und 76)

[Reynolds, 2002] Reynolds, J. C. (2002). Separation logic: a logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74. (zitiert auf Seite 77)

[Schaefer et al., 2010] Schaefer, I., Bettini, L., Bono, V., Damiani, F., and Tanzarella, N. (2010). *Delta-Oriented Programming of Software Product Lines*, pages 77–91. Springer Berlin Heidelberg, Berlin, Heidelberg. (zitiert auf Seite 9)

[Scholz et al., 2011] Scholz, W., Thüm, T., Apel, S., and Lengauer, C. (2011). Automatic detection of feature interactions using the java modeling language: An experience report. In *Proceedings of the 15th International Software Product Line Conference, Volume 2*, SPLC '11, pages 7:1–7:8, New York, NY, USA. ACM. (zitiert auf Seite 84)

http://dx.doi.org/10.24355/dbbs.084-201711280920

[Sedgewick and Wayne, 1983] Sedgewick, R. and Wayne, K. (1983). *Algorithms*. Addison-Wesley Professional. (zitiert auf Seite 85)

[Thüm et al., 2011] Thüm, T., Schaefer, I., Kuhlemann, M., and Apel, S. (2011). Proof composition for deductive verification of software product lines. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 270–277. (zitiert auf Seite 83)

[Thüm, 2015] Thüm, T. (2015). *Product-line specification and verification with feature-oriented contracts*. PhD thesis, Magdeburg, Universität, Diss., 2015. (zitiert auf Seite 2, 3, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 29, 33, 43, 45, 46, 51, 60, 70, 71, 72, 75, 77, 82, 83, 85, 86 und 87)

[Thüm et al., 2014] Thüm, T., Meinicke, J., Benduhn, F., Hentschel, M., von Rhein, A., and Saake, G. (2014). Potential synergies of theorem proving and model checking for software product lines. In *Proceedings of the 18th International Software Product Line Conference - Volume 1*, SPLC '14, pages 177–186, New York, NY, USA. ACM. (zitiert auf Seite 60, 61, 75, 84 und 85)

[Thüm et al., 2012] Thüm, T., Schaefer, I., Apel, S., and Hentschel, M. (2012). Family-based deductive verification of software product lines. In *Proceedings of the 11th International Conference on Generative Programming and Component Engineering*, GPCE '12, pages 11–20, New York, NY, USA. ACM. (zitiert auf Seite 60, 61, 75 und 84)

[Thüm et al., 2016] Thüm, T., Winkelmann, T., Schröter, R., Hentschel, M., and Krüger, S. (2016). Variability hiding in contracts for dependent software product lines. In *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems*, VaMoS '16, pages 97–104, New York, NY, USA. ACM. (zitiert auf Seite 61 und 75)

[Ullenboom, 2004] Ullenboom, C. (2004). *Java ist auch eine Insel*, volume 1475. Galileo Press. (zitiert auf Seite 30)

[Weigelt, 2013] Weigelt, A. (2013). Methoden-basierte komposition von kontrakten in feature-orientierter programmierung. *Bachelorarbeit, Universität Magdeburg, Germany, August*, pages 7–18. (zitiert auf Seite 84)

[Weiß, 2011] Weiß, B. (2011). *Deductive verification of object-oriented software: dynamic frames, dynamic logic and predicate abstraction*. KIT Scientific Publishing. (zitiert auf Seite 2, 12, 36, 37 und 82)

[Zhao and Rinard, 2003] Zhao, J. and Rinard, M. (2003). *Pipa: A Behavioral Interface Specification Language for Aspect*, pages 150–165. Springer Berlin Heidelberg, Berlin, Heidelberg. (zitiert auf Seite 2 und 76)

http://dx.doi.org/10.24355/dbbs.084-201711280920

http://dx.doi.org/10.24355/dbbs.084-201711280920

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Braunschweig, den 6. Oktober 2017

http://dx.doi.org/10.24355/dbbs.084-201711280920